Escuela Politécnica Superior

Descripción de Escenas para Robots Sociales mediante Deep Learning

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Álvaro Moreno Alberola

Tutor/es:
Miguel Ángel Cazorla Quevedo
Francisco Gómez Donoso

Septiembre 2018

Universitat d'Alacant
Universidad de Alicante

Preámbulo

"La fascinación y la gran curiosidad que provocan en mí las más nuevas tecnologías ha sido lo que me ha llevado, indudablemente, a decantarme por este proyecto. Desde que escuché hablar de la Inteligencia Artificial siempre he querido estudiarla en profundidad, conocer todas las técnicas posibles e inverstigar. Además, el hecho de que todo esto sea compatible con la robótica, tema que me ha llamado mucho la antención desde niño, ha hecho que mi interés creciese todavía más. ¿Cómo iba yo a negarme a dotar a un robot de inteligencia?

Dicho esto, se pueden intuir las bases del proyecto. En este caso se trata de hacer que un robot sea capaz de describir lo que ve. Cierto es que, a priori, un robot con esta capacidad no parece muy útil, pero si lees este proyecto te darás cuenta de lo interesante que puede llegar a ser"

Índice general

Índice de figuras

Índice de tablas

1 Introducción

Vivimos en una época dominada por la tecnología. Es una época en la que continuamente se descubren cosas nuevas, en la que la ciencia avanza a pasos agigantados, en la que cada vez más gente se suma al progreso, en la que la informática y el mundo de internet cobran cada vez más valor. Es una época en la que muchas de las empresas más importantes del mundo son tecnológicas, como Google, Apple o Microsoft. En esta época nada parece imposible. ¿Quién se creería hace una década que iba a poder conversar con su teléfono móvil como si fuese una persona más? ¿Quién hubiese dicho que los coches iban a conducirse por sí solos? ¿Cómo es posible que una máquina sea capaz de vencer al campeón del mundo de ajedrez?

Pues bien, este tipo de progresos de los que estoy hablando son hoy posibles gracias a la Inteligencia Artificial. En eso se va a centrar este proyecto. Se va a hacer uso de una de las ramas de la Inteligencia Artificial, el *Deep Learning*, para conseguir que un robot sea capaz de describir lo que ve. Se analizarán las diferentes técnicas mediante las cuales se puede resolver este problema y se implementará una de ellas.

Si alguien está leyendo esto, lo más probable es que ahora mismo se esté planteando si realmente esto tiene alguna utilidad. Puede que no sea algo que vaya a usarse en cada momento de la vida diaria de una persona, como las aplicaciones de mensajería de los *smartphones*. Pero un robot descriptor de escenas podría llegar a ser útil en muchas situaciones. Podría servir como asistente para una persona con problemas de visión. Podría utilizarse en escuelas para que los niños aprendan el nombre de los objetos, de hecho, estoy seguro que prestarían más atención a un robot que a cualquier profesor. Podría ser útil para obtener información de lugares a los que se no quiere o no se puede ir. Y como estos, se le podrían dar muchos más usos.

De todas formas, el descriptor de escenas es solo la parte final de este proyecto. Con esto quiero decir, que si no te resulta interesante este tipo de aplicación pero sí las tecnologías que se utilizan, puedes utilizar este proyecto como guía para conseguir reconocer objetos con un robot y darle cualquier otra utilidad. Es muy grande la cantidad de aplicaciones que se pueden desarrollar para un robot que reconoce objetos.

2 Marco Teórico

Las personas somos capaces de mirar cualquier escenario y reconocer los objetos que hay en él sin importar sus tamaños, sus posiciones, su color o cualquiera de sus características. Sin embargo, esta acción, que es tan simple para nosotros, se complica mucho si se quiere hacer mediante visión artificial. Es por eso que el principal problema al que se reduce este proyecto es al reconocimiento y localización de objetos.

Una vez conseguido esto, la descripción de escenas puede desarrollarse fácilmente. Habría que implementar unas cuantas reglas que trabajen con la información de los objetos para sacar conclusiones. Por eso, se van a estudiar distintas formas de reconocer objetos mediante visión artificial, de modo que se pueda decidir cual es la más óptima o interesante.

2.1. Métodos de reconocimiento de objetos

Hay muchas formas de abordar el problema de la detección de objetos, pero hoy en día las más populares son las relacionadas con *Machine Learning* y *Deep Learning*. Estas son las primeras que se van a estudiar.

2.1.1. Reconocimiento de objetos mediante Machine Learning

Antes de empezar a describir cómo usar este método para reconocer objetos, es conveniente asegurse de que se entiende lo que es el *Machine Learning*. Esta técnica consiste en el análisis de datos para enseñar a los ordenadores a aprender a partir de la experencia. Utiliza métodos computacionales para "aprender" información directamente desde los datos sin depender de una ecuación determinada como modelo. Este tipo de algoritmo es adaptable, mejorando su rendimiento cuando dispone de más muestras válidas de las que aprender.

Normalmente, los algoritmos de *Machine Learning* para reconocimiento de objetos funcionan de la siguiente manera:

1. Obtención de un buen conjunto de datos que sirva para el aprendizaje. Como se trata de reconocimiento de objetos, lo más común es que estos datos sean imágenes.

2. Extracción de características de las imágenes. Por ejemplo, algunas de las características que se suelen extraer son los bordes o esquinas de los objetos.

3. Procesamiento de estos datos con un modelo de *Machine Learning* que utilice las carácterísticas extraídas para separar los objetos en distintas clases.

4. Utilizar el modelo para analizar nuevas imágenes y clasificar nuevos objetos.

Esta imagen podría servir como resumen del proceso descrito anteriormente:

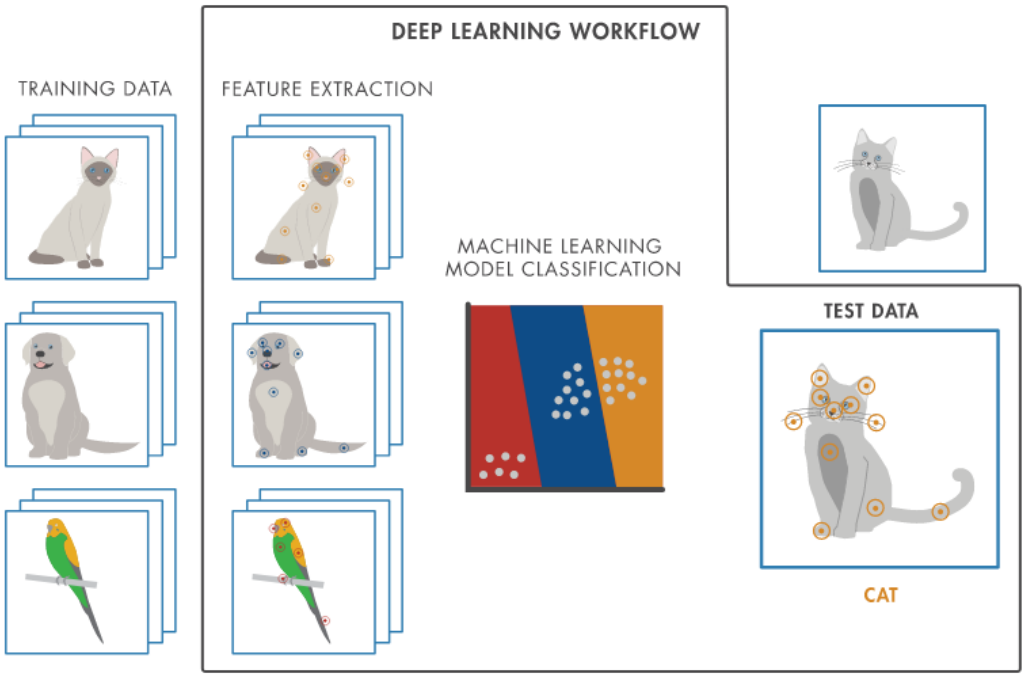


Figura 2.1: Proceso de reconocimiento de objetos mediante *Machine Learning*

Ventajas

Algunas de las ventajas más importantes:

- La existencia de muchos métodos de extracción de características y de muchos modelos de aprendizaje otorga flexibilidad. Siempre se pueden probar varias combinaciones hasta obtener un alto porcentaje de acierto.
- Se pueden llegar a conseguir buenos resultados con cantidades no muy grandes de datos.

Desventajas

Como desventajas podríamos decir:

- El propio algoritmo no extrae características, esto debe hacerse de forma "manual".
- Puede resultar complicado extraer características que identifiquen claramente la clase del objeto, ya que algunos objetos son similares entre sí.
- A veces es complicado obtener resultados muy precisos.

2.1.2. Reconocimiento de objetos mediante Deep Learning

Para que no haya confusión, es importante diferenciar entre *Machine Learning* y *Deep Learning*. El *Deep Learning* es un tipo de técnica incluida en el *Machine Learning*, que se basa en el aprendizaje a partir de ejemplos. Esta técnica permite que un modelo realice clasificaciones a partir de imágenes, textos o sonidos sin tener que extraer características primero. Estos modelos son entrenados usando grandes conjuntos de datos y arquitecturas de Redes Neuronales que contienen varias capas. Mediante *Deep Learning* se están consiguiendo resultados increíbles, llegando incluso a sobrepasar la precisión humana.

Existen dos formas de realizar reconocimiento de objetos mediante *Deep Learning*:

- **Entrenar un modelo desde cero**. Para ello es necesario disponer de un buen conjunto de datos, en el que cada muestra esté etiquetada con la clase a la que pertenece. Además, habría que diseñar una arquitectura de red neuronal que aprenda las características de cada tipo de objeto y construya el modelo.
- **Usar un modelo previamente entrenado**. Esto consiste en usar un modelo que ya haya sido entrenado con anterioridad para realizar las predicciones. También se puede utilizar una arquitectura ya creada y entrenarla con tu propio conjunto de datos. Volver a entrenar un modelo que ya ha sido entrenado puede hacer que el proceso se agilice.

En la siguiente imagen aparece el proceso interno de *Deep Learning* resumido:

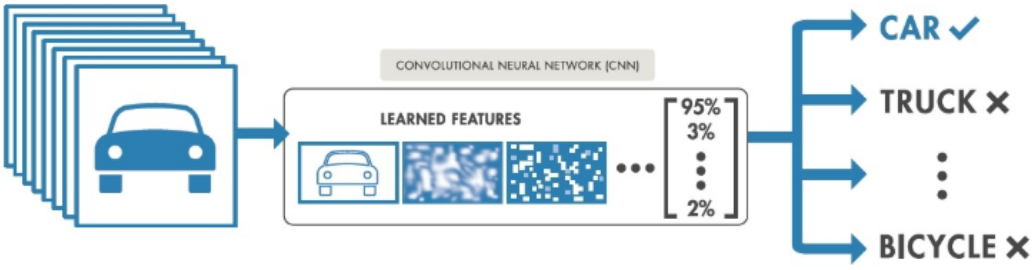


Figura 2.2: Proceso de reconocimiento de objetos mediante *Deep Learning*

Ventajas

Estas son las ventajas del *Deep Learning*:

- Los resultados que se obtienen suelen ser muy precisos, llegando incluso a superar a los que obtendría un humano.
- El propio algoritmo extrae las características de los objetos.
- Mediante el uso de modelos ya entrenados se pueden resolver problemas de forma muy rápida.

Desventajas

En cuanto a las desventajas, se podrían destacar las siguientes:

- Para que se realicen buenas predicciones hace falta un gran conjunto de datos que contenga muestras válidas. Además, estas muestras deben ser variadas, es decir, debe haber una cantidad similar de muestras de cada clase.
- Para poder entrenar modelos en un tiempo razonable hace falta disponer de una buena GPU, ya que la computación por CPU es bastante lenta.

2.1.3. Otros métodos

Las técnicas explicadas anteriormente son relativamente modernas. Por eso, antes de que adquirieran la popularidad que tienen ahora, se usaban otro tipo de aproximaciones para resolver problemas de este estilo. Algunas de estas aproximaciones son las siguientes:

Template Matching

Esta técnica consiste en el uso de imágenes pequeñas, o plantillas, para buscar regiones de una imagen más grande que coincidan con ellas. De esta forma, si se tuviese la plantilla de un coche, se podrían reconocer coches en otras imágenes.

Este método puede llegar a ser útil para problemas simples, sin embargo, pierde potencial para problemas de una embergadura mayor. Para el caso de una situación en la que se necesite reconocer una cantidad alta de objetos, habría que tener una cantidad alta de plantillas. Además, no siempre se pueden obtener buenos resultados, sobre todo cuando los objetos tienen un tamaño o forma poco habitual.

Segmentación de imágenes y reconocimiento de regiones

Este método consiste en la segmentación de imágenes en partes más pequeñas para analizarlas posteriormente. Se estudian las características de cada una de estas partes, como por ejemplo: el color, el tamaño, la sombra o el grosor de la línea. Este estudio exhaustivo de cada región puede servir para reconocer objetos.

Del mismo modo que la técnica anterior, esta puede ser útil para llevar a cabos tareas sencillas de reconocimiento, pero se queda corta para proyectos más grandes en los que hay que reconocer muchos tipos de objetos.

2.1.4. Aproximación elegida

El método elegido para llevar a cabo este proyecto ha sido el reconocimiento de objetos mediante *Deep Learning*. Las razones por las que se ha tomado esta decisión son las siguientes:

- Es el método más preciso de todos.
- Es una técnica muy moderna que se está utilizando mucho, y con mucho éxito, en la actualidad.
- Permite reconocer una cantidad grande de objetos.
- No es necesario trabajar sobre las imágenes para extraer sus características.
- El hecho de poder utilizar modelos ya entrenados agiliza y facilita mucho el proyecto.

2.2. Redes Neuronales Convolucionales

Puesto que se va a usar *Deep Learning* para el reconocimiento de objetos, es importante explicar uno de los tipos de algoritmos más importantes incluidos en esta técnica. Se trata de las *Convolutional Neural Networks* (CNN). Las CNN son un tipo de Redes Neuronales que resultan muy útiles para encontrar patrones en imágenes y reconocer objetos, caras y escenas. Aprenden directamente de las imágenes del conjunto de datos, usando patrones para clasificarlas.

El uso de este tipo de redes se ha vuelto muy popular debido a tres factores:

- Las propias redes extraen características por sí solas.
- Obtienen resultados muy precisos.
- Pueden ser reentrenadas con datos propios.

Una CNN puede tener decenas o cientos de capas, y cada una de ellas aprende a detectar diferentes características de una imagen. Se aplican filtros a cada una de las imágenes de entrenamiento con diferente resolución y la imagen que se obtiene como salida de una capa se usa de entrada para la siguiente. Los filtros pueden empezar siendo características muy simples, como el brillo o los bordes, e ir creciendo en complejidad, de forma que terminen siendo características muy concretas de un objeto.

Como las demás redes neuronales, las CNN están formadas por una capa de entrada, una capa de salida y otras capas intermedias, llamadas capas ocultas (*hidden layers*). Esta es la representación visual que se suele hacer de la estructura de las redes neuronales:

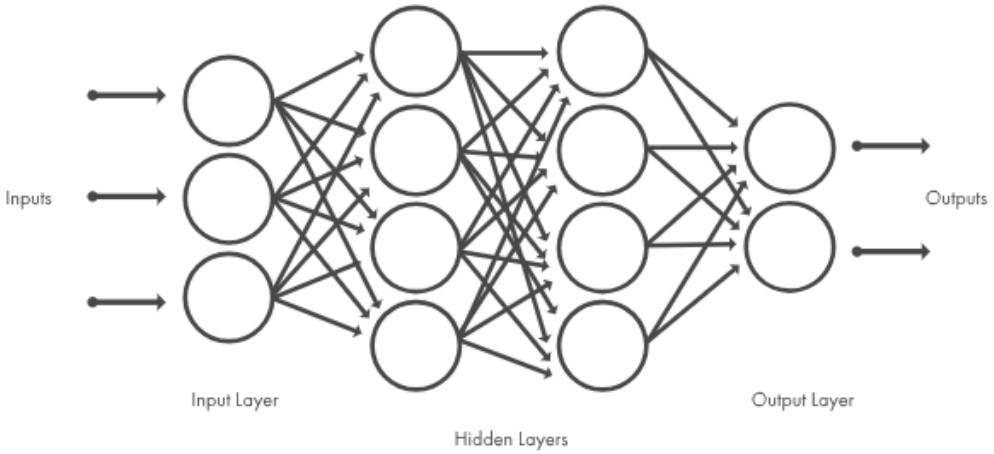


Figura 2.3: Estructura de una Red Neuronal

En estas capas se realizan operaciones que alteran los datos con la intención de aprender sus características. Cuando se usan Redes Convolucionales, las capas más comunes son las de **convolución** y ***pooling***. Las capas de convolución aplican algunos filtros convolucionales a las imágenes de entrada, activando cada filtro ciertas características. El *pooling* simplifica la salida, reduciendo la resolución de forma no lineal y reduciendo también el número de parámetros que la red necesita para aprender. Además de esto, las redes suelen aplicar unas funciones matemáticas, llamadas funciones de activación, que también trabajan sobre los datos. Este sería un ejemplo de cómo trabaja una red convolucional:

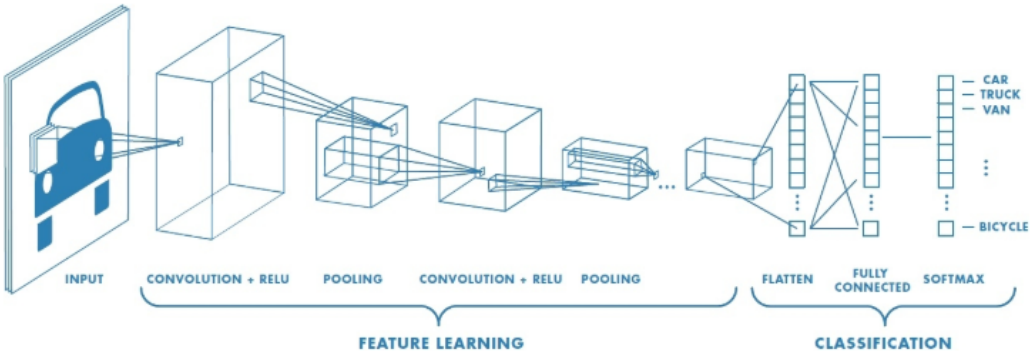


Figura 2.4: Ejemplo de funcionamiento de una Red Convolucional

Después de las capas que sirven para extraer características, la arquitectura cambia a capas de clasificación. La penúltima capa suele ser de tipo *fully-connected* (cada "neurona" se une con todas las de la siguiente capa), cuya salida es un vector que tiene tantos componentes como clases la red sea capaz de predecir. La última capa sirve para obtener la clasificación final.

2.2.1. Redes Neuronales Convolucionales de Región

A continuación se van a explicar los fundamentos de las *Region based Convolutional Neural Network* (R-CNN), ya que la red que se va a utilizar en este proyecto, *You Only Look Once* (YOLO), es de este tipo.

Las R-CNN son un tipo concreto de CNN basadas en la selección de regiones. Se utiliza un método de búsqueda selectivo, mediante el cual se extraen 2000 regiones de la imagen que se procese. De esta forma, el número de regiones de la imagen a clasificar se reduce en gran medida. La selección de regiones se hace utilizando el siguiente algoritmo:

1. Se genera una sub-segmentación inicial para obtener algunas regiones como candidatas.

2. Se utiliza un algoritmo voraz para combinar regiones parecidas y conseguir regiones más grandes.

3. Se usan estas regiones para producir las regiones finales.

Este podría ser un ejemplo de funcionamiento de un algoritmo de este tipo:

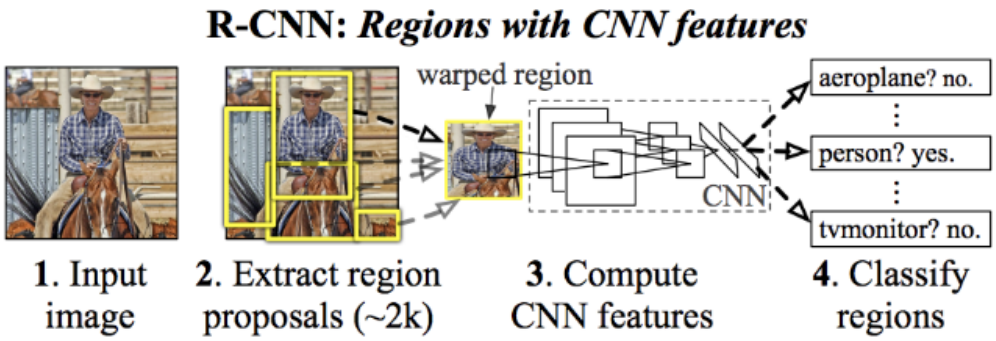


Figura 2.5: Selección de regiones en una Red Convolucional de Región

Las 2000 regiones elegidas se combinan en forma de cuadrado y se le pasan a una CNN que produce un vector de características de 4096 dimensiones. La CNN se encarga de extraer las características de la imagen, que serán pasadas a un algoritmo de clasificación. Este, a su vez, determinará la presencia de objetos en cada una de las regiones. Además, este tipo de redes no solo predice la presencia de objetos, si no que también los localiza y los enmarca con *bounding boxes*.

3 Objetivos

Hablando en términos generales, el objetivo de este proyecto ya se ha comentado anteriormente: conseguir que un robot sea capaz de describir las imágenes que capta con su cámara. Pero esto es un objetivo global, que hace ver este trabajo de forma muy abstracta. Además, es un objetivo que no puede cumplirse de forma inmediata, sino que es necesario dividirlo en una serie de tareas o sub-objetivos para llevarlo a cabo. Estos sub-objetivos dan información más específica sobre lo que se debe hacer y podrían ser los siguientes:

- **Aprender a usar *Robot Operating System* (ROS)**. Puesto que este es un proyecto en el que la robótica juega un papel muy importante, es de mucha utilidad usar herramientas que faciliten la obtención de información y la comunicación con el robot. Por esta razón se ha decidido usar ROS, el framework más utilizado en este ámbito. Sin embargo, no es sencillo trabajar con ROS, por eso, uno de los primeros objetivos para poder completar el proyecto es aprender a utilizarlo.

- **Obtener conocimientos sobre *Deep Learning***. Antes de empezar a desarrollar una solución para este proyecto, hay que conocer bien las técnicas que se van a utilizar, de modo que se les pueda sacar el máximo partido posible. Por tanto, otro de los objetivos que deben cumplirse a corto plazo es la investigación de este campo de la IA. Cabe destacar que este tema es muy profundo y que el proyecto estará centrado principalmente en un solo tipo de aprendizaje, uno mediante Redes Neuronales, más concrectamente, CNN.

- **Saber preparar el equipo para un proyecto de estas características**. Para llevar a cabo este proyecto, primero hay que configurar el entorno de trabajo para que sea el adecuado. Y es que, para hacerlo funcionar, es necesario realizar una serie de instalaciones que serán explicadas más adelante. Estos son los componentes necesarios:

  - Python
  - ROS
  - Nvidia Drivers
  - CUDA Toolkit
  - OpenCV
  - Darknet

- **Implementar un reconocedor de objetos**. Este es un paso previo al descriptor de escenas. Antes de poder descibir los objetos de una imagen, primero hay que identificarlos. Esta funcionalidad no es solo aplicable a robótica, sino que podría usarse en muchos tipos de aplicaciones distintas.
- **Aprender a usar librerías de tratamiento de imágenes como OpenCV**. Esto es necesario para poder trabajar sobre las imágenes una vez reconocidos los objetos, de forma que el proyecto quede más visual y más claro.

4 Metodología

En este apartado se explicarán todos los instrumentos, herramientas y técnicas que se han usado para llevar a cabo este proyecto. Desde la información sobre el robot que se utiliza, hasta las instalaciones de las herramientas usadas para darle la funcionalidad deseada

4.1. Robot Pepper

El robot Pepper es un robot humanoide creado por la empresa "Aldebaran Robotics" para ser el sucesor del robot NAO. Uno de los principales cambios que presenta Pepper respecto a NAO es la ausencia de piernas, ya que esta nueva versión cuenta con tres ruedas omnidireccionales para poder transportarse de un sitio a otro. Además, a Pepper se le ha incluido un motor emocional que le permite estudiar los gestos y el rostro de las personas que interactúan con él. De esta forma, puede tomar decisiones durante la conversación que hagan sentir mejor a la persona.

Estas nuevas características y algunas otras hacen que Pepper esté creando grandes expectativas en el mundo de la robótica. A continuación, se detallan algunas de las características que este incluye.

Especificaciones técnicas

- **Peso**: 28 kg.
- **Altura**: 120 cm.
- **Fondo**: 42,5 cm.
- **Batería**: Litio, 30,0Ah/795Wh.
- **Autonomía**: Hasta 12 horas.
- **Conectividad**: Wi-Fi / Ethernet.
- **Velocidad**: Más de 3km/h
- **Motores**: 20.
- **Partes móviles**: Cabeza (1), hombros (2), codos(2), muñecas(2), dedos(10), caderas (1) y rodillas (1).

- **Ruedas**: 3 (omnidireccionales).
- **Movimiento**: 360º.
- **Velocidad máxima**: 3 km/h.
- **Tablet**: LG CNS.

En la siguiente imagen se pueden apreciar algunas de estas características y se puede ver la cantidad de dispositivos que incluye el robot Pepper:

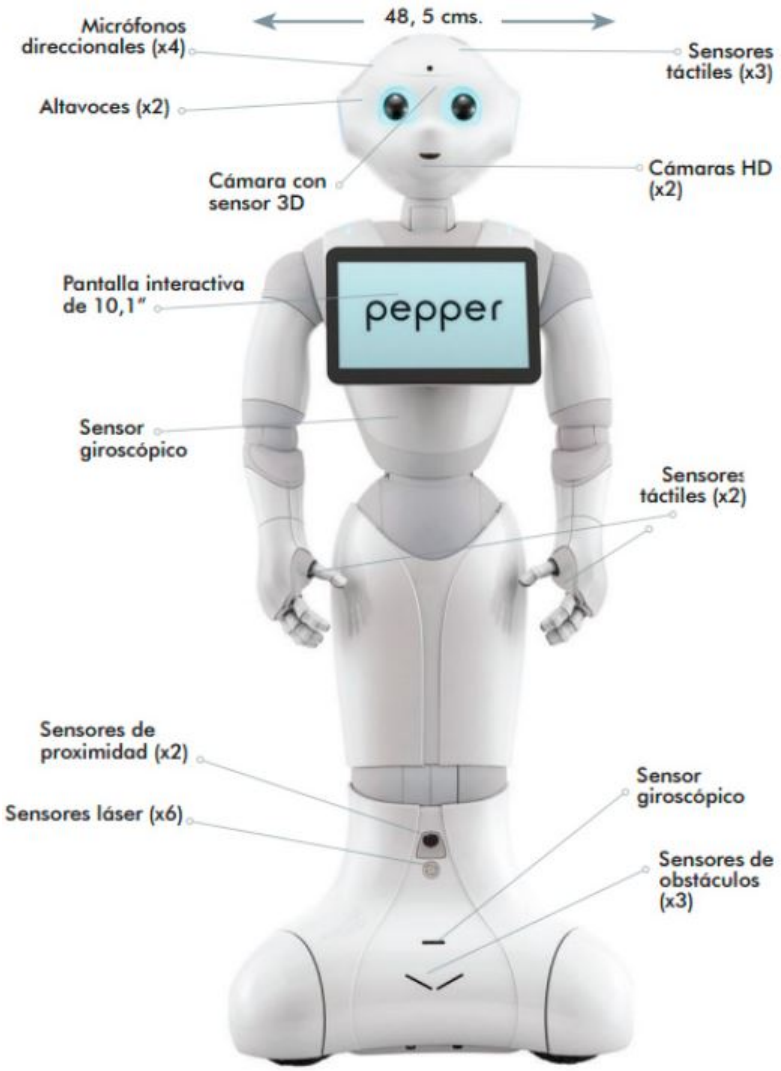


Figura 4.1: Robot Pepper

4.2. ROS

Puesto que nunca antes había trabajado con robots ni había hecho ningún proyecto relacionado con la robótica, antes de empezar a desarrollar nuevas aplicaciones para Pepper, o incluso antes de tener un objetivo claro en este Trabajo Final de Grado (TFG), he tenido que aprender a utilizar una herramienta que es fundamental para llevar a cabo tareas de este tipo. En este caso, la herramienta de la que estoy hablando es la librería *Robot Operating System* (ROS) compatible con Ubuntu.

Para formarme en el uso de esta librería, se me proporcionó un tutorial cuyo objetivo es hacer que un robot se desplace por un conjunto de habitaciones simuladas sin colisionar con ninguna pared. A continuación, se explican los pasos seguidos para completar este tutorial de aprendizaje de ROS con éxito.

4.2.1. Instalación

Como es lógico, antes de empezar con cualquier tarea se debe tener instalada y funcionando la librería mencionada anteriormente. Para ello, se ha entrado en la página web de ROS [ROS, 2018] y se han seguido los pasos descritos para la instalación.

ROS dispone de varias versiones y se debe elegir entre una u otra según la distribución de Ubuntu que se esté utilizando. En mi caso, puesto que utilizo la versión de Ubuntu 16.04, he instalado **ROS Kinetic**. Para ello, he hecho lo siguiente:

1. Preparar Ubuntu para que acepte la instalación de software proveniente de ROS mediante el comando:

Listado 4.1: Configuración de la aceptación de paquetes ROS

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
```

2. Configurar las claves para la descarga de paquetes:

Listado 4.2: Configuración de las claves de descarga

```
sudo apt-key adv --keyserver
hkp://ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

3. Elegir el tipo de instalación que se desea realizar. En mi caso, he elegido la instalación completa para evitar que una vez descargado falte alguna funcionalidad necesaria. Para realizar esta instalación se debe introducir el siguiente comando:

Listado 4.3: Descarga e instalación de ROS

```
sudo apt-get install ros-kinetic-desktop-full
```

4. Una vez terminado el proceso de instalación es necesario realizar algunos pasos más antes de poder utilizar ROS. Uno de estos pasos es la inicialización de rosdep y la actualización del mismo:

Listado 4.4: Inicialización y actualización de rosdep

```
sudo rosdep init
rosdep update
```

5. Finalmente, es conveniente configurar el entorno de trabajo haciendo que las variables de entorno estén listas cada vez que se abra una terminal. Para ello, las agregamos al archivo *bashrc*:

Listado 4.5: Configuración del entorno

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

4.2.2. Conceptos básicos

Una vez instalado y antes de porfundizar en el tutorial de aprendizaje, es importante tener claros algunos conceptos claves en el funcionamiento de ROS. Estos harán que entendamos la forma en la que se debe trabajar con la librería y cómo funciona internamente. Los conceptos más importantes a tener en cuenta son los siguientes:

- **Nodo**: Un nodo es un proceso que realiza cálculos y tareas. Los nodos pueden combinarse entre ellos mediante el uso de tópicos y otros medios.
- **Tópico**: Los tópicos pueden definirse como canales de información entre los nodos. En ellos la comunicación funciona de forma unidireccional, ya que los nodos pueden suscribirse a ellos pero desde el tópico no se controla qué nodos están suscritos.
- **Paquete**: Un paquete es un contenedor en cuyo interior podemos encontrar un nodo, una librería, un conjunto de datos o cualquier cosa que pueda constituir un módulo. Mediante ellos está organizado el software de ROS.
- **Pila**: Una pila es un conjunto de nodos que, juntos, proporcionan alguna funcionalidad.

4.2.3. Comandos de ROS

A cotinuación, se explican los comandos principales que se usarán a lo largo de todo este proyecto:

- **roscore**: Pone en marcha todo lo necesario para dar soporte de ejecución al sistema completo de ROS. Siempre tiene que estar ejecutándose para permitir que los nodos se comuniquen entre sí.
- **roscd**: Cambia a un directorio de paquete o pila.
- **rosnode**: Permite obtener información sobre los nodos y manejarlos. Hay diversas modalidades de este comando:
  - **rosnode list**: Muestra los nodos en ejecución.
  - **rosnode info nombre_nodo**: Muestra la información del nodo.
  - **rosnode kill nombre_nodo**: Mata el proceso correspondiente con el nodo.
- **rosrun**: Permite ejecutar cualquier aplicación de un paquete sin necesidad de cambiar a su directorio.
- **rostopic**: Permite obtener información sobre los nodos y manejarlos. Hay diversas modalidades de este comando:
  - **rostopic list**: Imprime información sobre los tópicos activos.
  - **rostopic info**: Imprime información de un tópico.
  - **rostopic pub**: Publica información a un tópico activo.
- **rqt_graph**: Permite visualizar los nodos que se están ejecutando y el paso de tópicos entre ellos.

4.2.4. Tutorial de programación en ROS

Una vez instalado ROS y entendidos los conceptos básicos, ya es posible completar el tutorial de programación. Mediante el tutorial, es posible realizar un primer acercamiento a la programación usando este framework y a los comandos descritos anteriormente. En él se utiliza el simulador **Stage**, que servirá para realizar las pruebas deseadas en un robot móvil en 2D.

Lo primero que se debe hacer para empezar a utilizar ROS es ejecutar el comando:

Listado 4.6: Inicio de ROS

```
roscore
```

El siguiente paso es la ejecución de Stage. Para ello se necesitan dos ficheros: uno que contiene la configuración del mundo que se simulará y otro que contiene la imagen del mapa de este mundo. Estando estos dos ficheros descargados, hay que ir al directorio donde estén ubicados y ejecutar el comando:

Listado 4.7: Lanzamiento de Stage

```
rosrun stage_ros stageros ra1.cfg
```

Se puede observar como inmediatamente se abre el mundo simulado mostrando la imagen descargada.

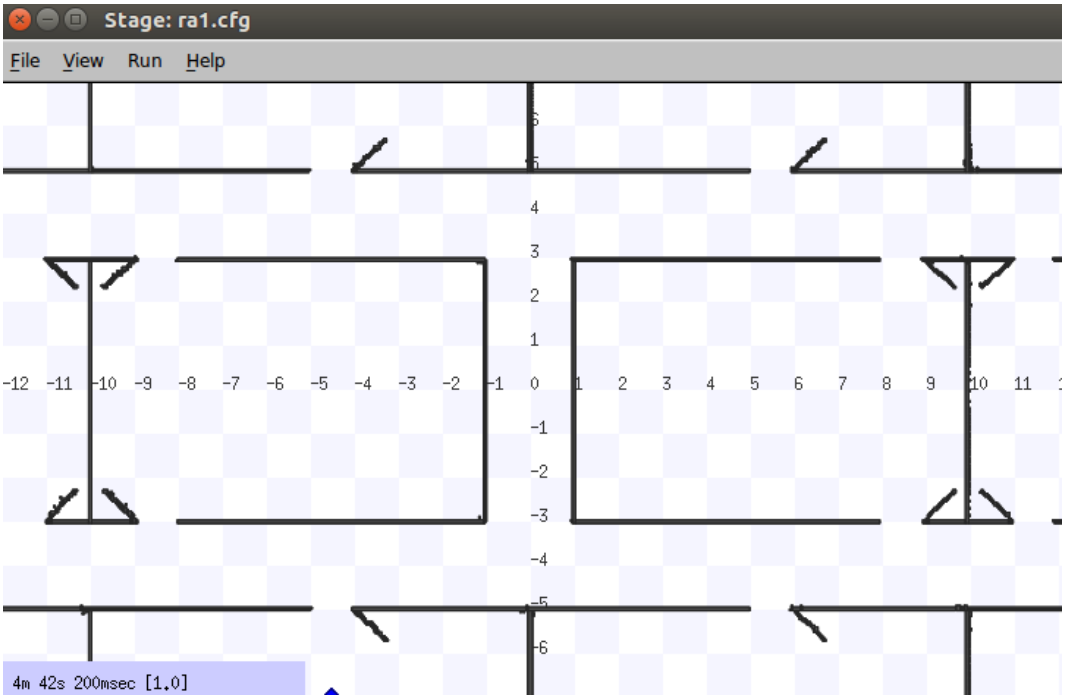


Figura 4.2: Ejecución del simulador Stage

Lo primero que se debe hacer para probar el simulador es mover al robot teleoperadamente, mediante teclado. Esto se hará mediante un nodo de ROS ya creado. Para utilizarlo, se debe crear un directorio donde estarán todos los paquetes y nodos que creemos. A este directorio se le da el nombre **catkin_ws**. Dentro de este directorio hay que crear otro con el nombre **src** y ejecutar el comando:

Listado 4.8: Inicialización de Catkin

```
catkin_init_workspace
```

Nuevamente, dentro del directorio src, hay que lanzar otro comando para descargar el paquete ROS de teleoperado:

Listado 4.9: Descarga del paquete de teleoperado

```
git clone https://github.com/ros-teleop/teleop_twist_keyboard.git
```

Una vez hecho esto, el paquete ya está listo para usarse. Hay que introducir los comandos de compilación y ejecución desde el directorio catkin_ws. Estos comandos son:

Listado 4.10: Ejecución del teleoperado por teclado

```
catkin_make
source devel/setup.bash
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

El segundo comando de los tres mencionados anteriormente no es realmente de compilación o ejecución. Su función es transmitirle a nuestro sistema que en el directorio catkin_ws hay ejecutables ROS.

Introducidos estos tres comandos, el robot se podrá controlar siguiendo las instrucciones que se muestran por pantalla:

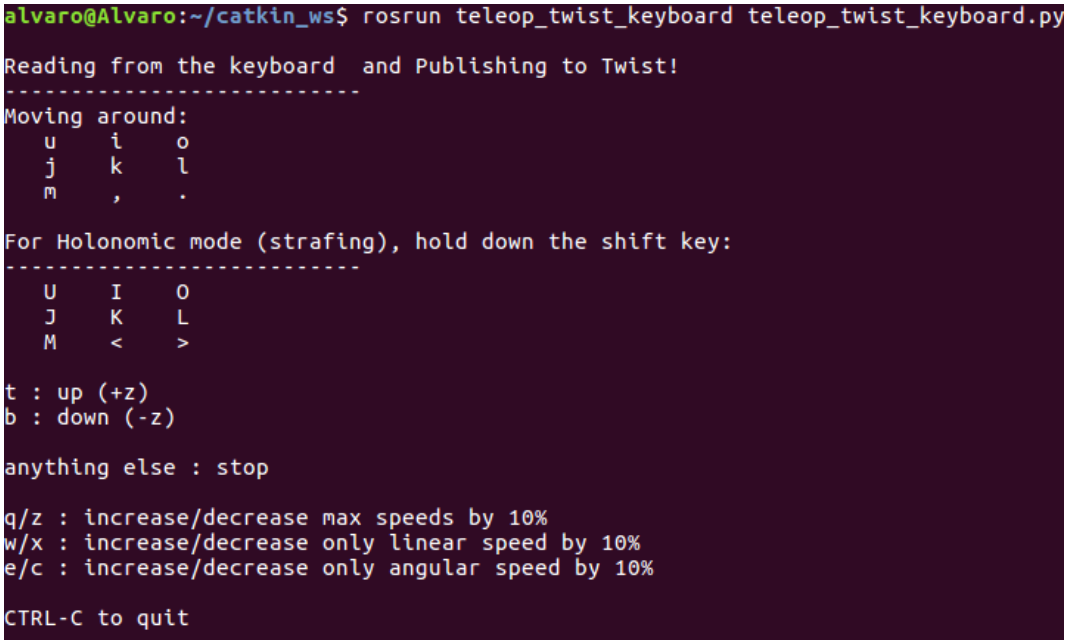


Figura 4.3: Comandos de teleoperado

El siguiente paso en este tutorial es hacer que el robot pueda moverse de forma autónoma. Para hacer esto, se necesitará un nuevo nodo, aunque esta vez va a ser creado en vez de descargado. Antes de crear el nodo, se ha de crear un paquete (**wander**) para que lo contenga. Para ello se usa el siguiente comando, que crea el paquete con todas sus dependencias:

Listado 4.11: Creación el paquete wander

```
catkin_create_pkg wander roscpp geometry_msgs sensor_msgs nav_msgs
```

Este nodo va a ser programado en `C++`, por eso es necesario modificar el archivo **Makefile** del paquete para que se cree un ejecutable cada vez que es compilado. Estas

son las lineas que se deben añadir:

Listado 4.12: Modificación de los parámetros de compilación

```
add_executable(${PROJECT_NAME} src/wander.cpp)
target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES} )
```

Hecho esto, hay que descargar el archivo de código que será el que implemente el nodo y meterlo dentro de la carpeta **src** del paquete.

Lo único que hace el código descargado es implementar un nodo ROS que publica mensajes de velocidad. Inicialmente, el nodo está programado para que el robot vaya en línea recta. Para completar el tutorial, hay que conseguir que este no choque con las paredes. Para ello, lo que se ha hecho ha sido acceder a la información que proporciona el láser del robot y hacer que este gire cuando detecte un obstáctulo a una distancia determinada con el haz frontal del láser. Estas son las lineas de código que implementan esta parte:

Listado 4.13: Implementación de la navegación

```
1  int valorMedio;
2
3  if (totalValues %2 == 0){
4          valorMedio = totalValues / 2;
5  }
6  else{
7          valorMedio = (totalValues + 1) / 2;
8  }
9  int valorIzq = valorMedio / 2;
10 int valorDer = valorMedio + valorIzq;
11 if(msg->ranges[valorMedio] < 2 && msg->ranges[valorMedio] > msg->
       range_min || (msg->ranges[valorIzq] < 0.5 || msg->ranges[valorDer]
       < 0.5)){
12   rotateVel = 1;
13   forwardVel = 0;
14 }
15 else{
16   forwardVel = 4;
17   rotateVel = 0;
18 }
```

Hecho esto, el objetivo del tutorial estaría cumplido. Cabe destacar que este tutorial hace que las primeras impresiones sobre ROS sean buenas y facilita la introducción en la herramienta.

4.3. Darknet

4.3.1. Introducción a Darknet

Darknet es un framework para Redes Neuronales de código abierto que está escrito en C y CUDA. Es rápido y fácil de instalar, además, permite la computación tanto por *Central Processing Unit* (CPU) como por *Graphics Processing Unit* (GPU). Este es el símbolo representativo de Darknet:

Figura 4.4: Logo de Darknet

Este framework ha sido desarrollado por Joseph Redmon, un hombre que se define a sí mismo como:

"I am a computer scientist with a love of machine learning, data analysis, and low level program design and implementation. Outside of computer science, I enjoy skiing, hiking, rock climbing, and playing with my Alaskan malamute puppy, Kelp."

Dejando de lado sus aficiones, se puede decir que Joseph Redmon es un hombre que dedica su vida a la ciencia computacional y que, con este framework, ha hecho grandes avances en el mundo del Machine Learning y las Redes Neuronales.

En este proyecto se van a dejar de lado muchas de las funciones que integra Darknet, centrando la atención principalmente en YOLO, un tipo R-CNN que será explicada a continuación. Esta es una lista de las otras herramientas que incluye Darknet:

- **ImageNet Classification**: Sirve para clasificar imágenes mediante algunas arquitecturas ya conocidas como AlexNet o VGG-16.
- **Nightmare**: Este es otro proyecto basado en Redes Neuronales, más concretamente en CNN, orientado a la aplicación de estas redes a algunos aspectos del pasado.

- **RNN's**: Esta es otra de las funcionalidades que tiene Darknet y es que este framework aplica CNN para generar lenguaje y trabajar el procesamiento de lenguaje natural.
- **DarkGo**: DarkGo es otro proyecto incluido en Darknet que consiste en la aplicación de redes neuronales en el juego AlphaGo para la predicción de movimientos.
- **Tiny Darknet**: Es un proyecto en el que se implementa una red neuronal "pequeña". Esto quiere decir que el modelo que genera es menos pesado que los que generan otras arquitecturas. Además, hace un número mucho menor de operaciones. En esta tabla pueden verse los datos exactos:

MODELO	OPERACIONES	TAMAÑO
AlexNet	2.27 Bn	238 MB
Darknet Reference	0.81 Bn	28 MB
SqueezeNet	2.17 Bn	4.8 MB
Tiny Darknet	0.98 Bn	4.0 MB

Tabla 4.1: Comparación de modelos de CNN

- **Classifier on CIFAR-10**: Esto es un tutorial de cómo entrenar un clasificador para Darknet desde cero.

4.3.2. Instalación básica

Esta instalación no permite hacer uso de todas las funcionalidades de YOLO, ni siquiera es suficiente para poder ejecutar la aplicación final de este proyecto. Solo permitirá realizar predicciones sobre imágenes usando la CPU y guardarlas en el disco. Para usar YOLO de forma completa, se debe ir a la instalación avanzada, explicada más adelante.

Para instalar Darknet solo hay que descargar el proyecto de GitHub y compilarlo. Se puede conseguir siguiendo estos pasos:

1. Para descargar el proyecto hay que introducir el comando:

Listado 4.14: Descarga de Darknet

```
git clone https://github.com/pjreddie/darknet
```

2. Para compilarlo solo hay que ir al directorio recién descargado llamado **darknet** y usar el comando **Make**:

Listado 4.15: Compilación de Darknet

```
cd darknet
make
```

Hecho esto y comprobado que la compilación se lleve a cabo sin errores, Darknet estaría instalado en el equipo. Una compilación correcta debe tener una salida como la siguiente:

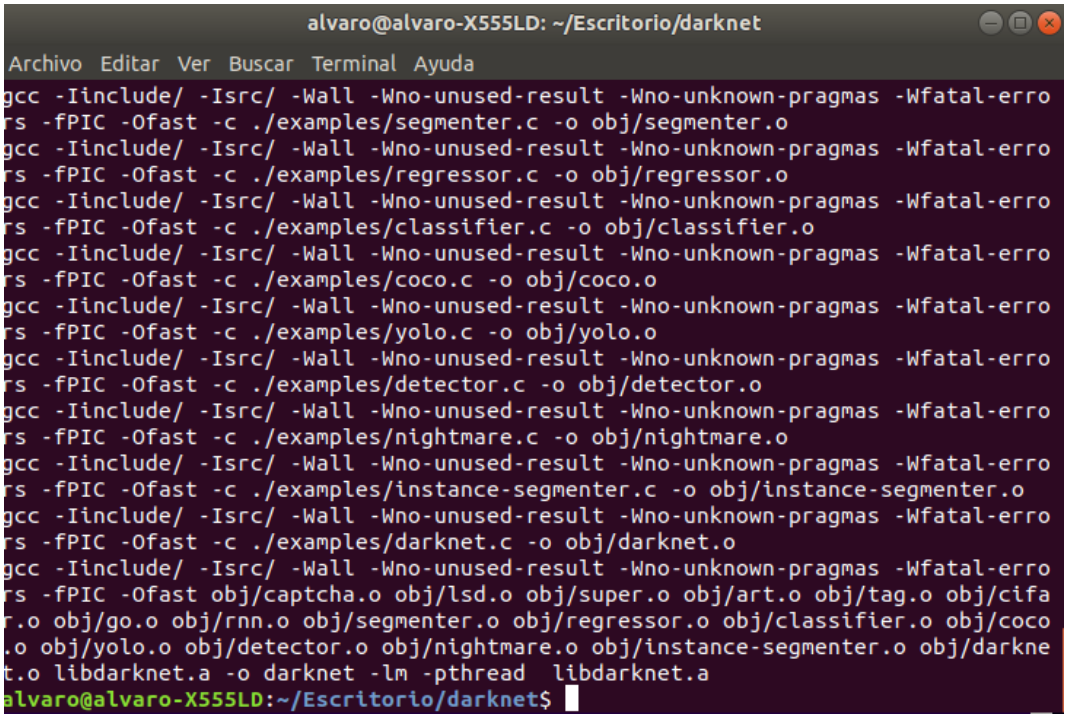


Figura 4.5: Compilación de Darknet

4.3.3. Instalación de dependencias

Antes de poder instalar Darknet de forma avanzada, es necesario instalar algunas dependencias necesarias para su funcionamiento. Estas dependencias son las siguientes:

- Nvidia Drivers
- CUDA Toolkit
- OpenCV

A continuación, se explica cómo instalar cada uno de estos componentes.

Instalación de los drivers de Nvidia

La instalación de los *drivers* de Nvidia es un proceso sencillo, ya que puede completarse introduciendo unos cuantos comandos en terminal. Estos son los pasos a seguir:

1. Antes de empezar con la instalación, es importante saber qué versión de los drivers es la que se debe instalar. Esto se puede saber fácilmente accediendo a la página de Nvidia [nvi, 2018] e introduciendo el modelo de la tarjeta gráfica. En mi caso la GPU es la siguiente:

Figura 4.6: Datos de mi tarjeta gráfica

Y los *drivers* compatibles con esta GPU:

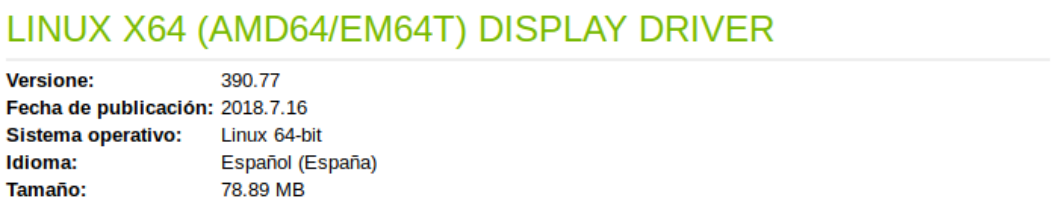


Figura 4.7: Versión de los *drivers* de Nvidia

2. El primer paso de la instalación es añadir el repositorio de descarga a la lista de repositorios del equipo, de forma que puedan descargarse los paquetes deseados.

Listado 4.16: Configuración del repositorio de descarga

```
sudo add-apt-repository ppa:graphics-drivers
```

3. A continuación, se debe actualizar el sistema para que tenga en cuenta el repositorio añadido.

Listado 4.17: Actualización del sistema

```
sudo apt-get update
```

4. Para terminar la instalación, hay que introducir un comando más en el que se debe elegir la versión deseada de los *drivers* de Nvidia. En mi caso, la versión es la 390.77, por lo que el comando introducido fue el siguiente:

Listado 4.18: Instalación de los *drivers* de Nvidia

```
sudo apt-get install nvidia-390
```

5. Una ver terminada la instalación se debe reiniciar el equipo. Se puede comprobar que todo ha salido correctamente con el comando:

Listado 4.19: Comprobación de la versión de los *drivers* de Nvidia

```
nvidia-smi
```

Si la instalación es correcta el *output* del comando debería ser parecido el siguiente:

```
alvaro@alvaro-Z270-HD3P:~$ nvidia-smi
Sat Aug 25 10:37:20 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.48                 Driver Version: 390.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 106...  Off  | 00000000:01:00.0  On |                  N/A |
|  0%   38C    P0    37W / 180W |    182MiB /  3018MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1036      G   /usr/lib/xorg/Xorg                           138MiB |
|    0      2568      G   compiz                                        41MiB |
+-----------------------------------------------------------------------------+
```

Figura 4.8: Comprobación de la versión de los *drivers* de Nvidia

Instalación de CUDA

Este paso es incluso más fácil que el anterior, ya que sobra con un único comando de instalación:

Listado 4.20: Instalación de CUDA

```
sudo apt-get install nvidia-cuda-toolkit
```

Cuando el comando termina de ejecutarse es posible comprobar la instalación introduciendo en terminal:

Listado 4.21: Comprobación de la versión de CUDA

```
nvcc -V
```

Si todo ha ido bien deberían aparecer algunos datos de esta herramienta como, por ejemplo, la versión. La salida debería ser algo parecido a esto:

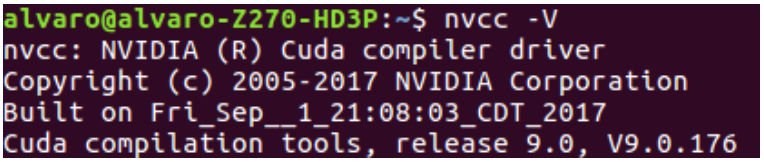


Figura 4.9: Comprobación de la versión de CUDA

Instalación OpenCV

La instalación de OpenCV es algo más compleja debido al gran número de dependencias que tiene. Estos son los pasos seguidos para llevarla a cabo:

1. Lo primero es asegurarse de que el sistema está totalmente actualizado. Para ello hay que introducir los comandos:

Listado 4.22: Actualización del sistema

```
sudo apt-get -y update
sudo apt-get -y upgrade
sudo apt-get -y dist-upgrade
sudo apt-get -y autoremove
```

2. A continuación, hay que instalar todas las dependencias de OpenCV. Una dependencia es un paquete que debe estar instalado en el sistema para que OpenCV pueda funcionar sin errores. Esta es la lista de comandos que se deben introducir para instalar todas las dependencias necesarias:

Listado 4.23: Instalación de dependencias de OpenCV

```
sudo apt-get install -y build-essential cmake

sudo apt-get install -y qt5-default libvtk6-dev

sudo apt-get install -y zlib1g-dev libjpeg-dev libwebp-dev libpng-dev
    libtiff5-dev libjasper-dev libopenexr-dev libgdal-dev

sudo apt-get install -y libdc1394-22-dev libavcodec-dev libavformat-dev
    libswscale-dev libtheora-dev libvorbis-dev libxvidcore-dev libx264-dev
    yasm libopencore-amrnb-dev libopencore-amrwb-dev libv4l-dev libxine2-dev

sudo apt-get install -y libtbb-dev libeigen3-dev
```

```
sudo apt-get install -y python-dev python-tk python-numpy python3-dev
    python3-tk python3-numpy

sudo apt-get install -y ant default-jdk

sudo apt-get install -y doxygen
```

3. El siguiente paso es instalar OpenCV, siempre y cuando la instalación de las dependencias se complete sin problemas. Esta herramienta no dispone de un instalador para distribuciones Linux, sino que hay que descargar los fuentes y compilarlos. Para descargar los fuentes:

Listado 4.24: Descarga de OpenCV

```
wget https://github.com/opencv/opencv/archive/3.4.0.zip
```

El último número de la url es la versión de OpenCV que se desea instalar. En mi caso, la 3.4.0. Una vez descargado el arhcivo podemos descomprimirlo de la siguiente forma:

Listado 4.25: Descompresión de OpenCV

```
unzip 3.4.0.zip
```

O simplemente haciendo click derecho sobre el archivo y seleccionando "**extraer aquí**". Cuando se haya completado la extracción, hay que ir al directorio extraído y crear una carpeta para la compilación:

Listado 4.26: Creación de la carpeta de compilación

```
cd opencv-3.4.0
mkdir build
```

Hecho esto, se debe ir a la carpeta recién creada

Listado 4.27: Cambio al directorio de compilación

```
cd build
```

e introducir los siguientes comandos para realizar la compilación e instalación:

Listado 4.28: Compilación e instalación de OpenCV

```
cmake -DWITH_QT=ON -DWITH_OPENGL=ON -DFORCE_VTK=ON -DWITH_TBB=ON -DWITH_GDAL
    =ON -DWITH_XINE=ON -DBUILD_EXAMPLES=ON -DENABLE_PRECOMPILED_HEADERS=OFF
    ..

make -j4

sudo make install
```

```
sudo ldconfig
```

Este es un proceso largo que puede requerir de bastante tiempo para completarse.

4. Finalmente, es recomendable comprobar que la instalación se haya llevado a cabo de forma correcta. Hay diversas formas de hacer esto, en mi caso lo que hago es abrir un *prompt* de Python escribiendo en terminal

Listado 4.29: Apertura de un prompt de Python

```
python
```

e introducir lo siguiente para comprobar la versión de OpenCV:

Listado 4.30: Comprobación de la versión de OpenCV

```
import cv2 as cv
cv.__version__
```

Tal y como se aprecia en la imagen:

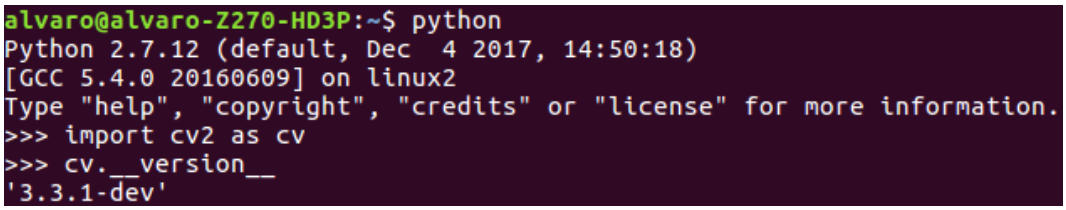


Figura 4.10: Comprobación de la versión de OpenCV

4.3.4. Instalación avanzada

Este tipo de compilación es la necesaria para poder completar el proyecto. Está basada en las herramientas CUDA y OpenCV, instaladas anteriormente.

De la misma forma que en la instalación básica, se empieza por descargar Darknet mediante el comando mostrado en el listado 4.14. Una vez descargado, habría que modificar las primeras líneas del archivo **Makefile** ubicado en el directorio de Darknet, dejándolas así:

Listado 4.31: Cambio en el archivo Makefile de Darknet

```
1 GPU=1
2 CUDNN=0
3 OPENCV=1
```

```
4 OPENMP=0
5 DEBUG=0
```

Como se puede intuir, esto sirve para activar la compilación usando CUDA y OpenCV. Hecho esto, solo habría que compilar Darknet usando el comando mostrado en el listado 4.15. Si la compilación se lleva a cabo sin errores, el *output* debería ser parecido a este:

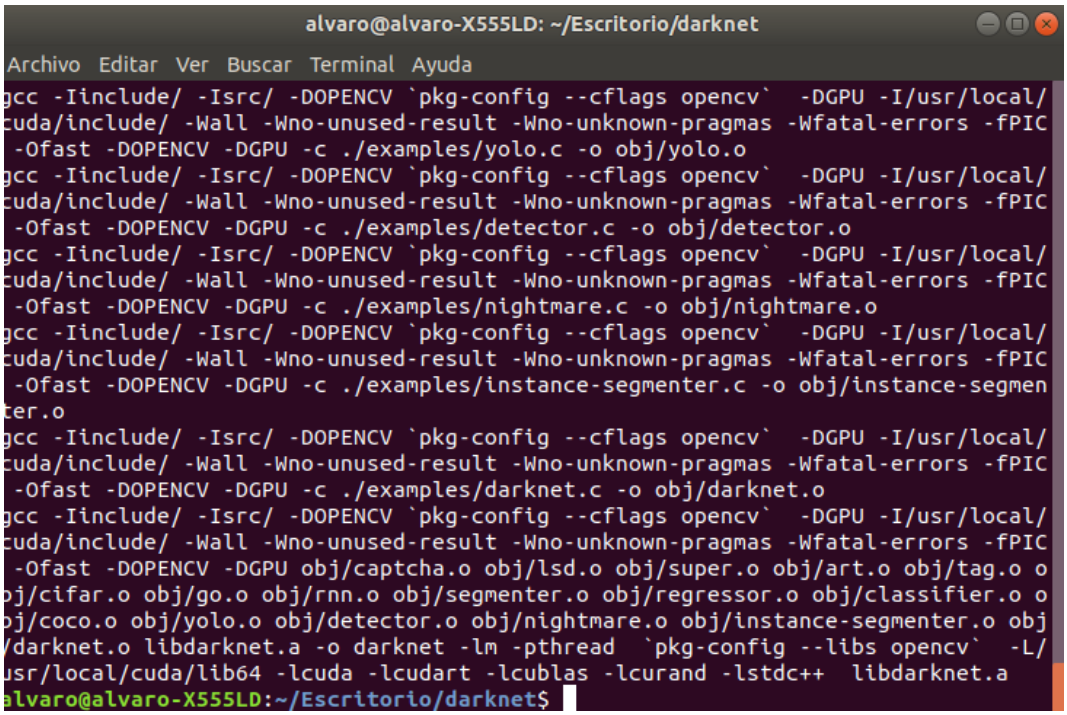


Figura 4.11: Compilación avanzada de Darknet

5 Cuerpo

En este apartado se presentarán y analizarán los resultados obtenidos con este proyecto y se explicará cómo llegar hasta ellos.

5.1. YOLO

5.1.1. Introducción

YOLO es un sistema de detección de objetos en tiempo real basado en *Deep Learning*. El nombre proviene de las siglas "*You Only Look Once*", cuya traducción literal es "solo miras una vez". Esto quiere hacer referencia a la rapidez con la que el sistema es capaz de reconocer objetos. Joseph Redmon define este sistema de la siguiente forma:

"*You Only Look Once* (YOLO) *is a state-of-the-art, real-time object detection system. On a Pascal Titan X it processes images at 30 FPS and has a mAP of 57.9 % on COCO test-dev.*"

Con esto quiere decir que YOLO es lo último en sistemas de detección de objetos. En una Pascal Titan X procesa imágenes a 30 FPS (Fotogramas Por Segundo) y tiene un mAP del 57.9 % en el dataset COCO.

Además de esta definición, en la página de Darknet se puede ver un vídeo [dem, 2018] donde se aprecia el potencial que tiene realmente YOLO. El autor también adjunta un gráfico donde se compara YOLO con otros sistemas similares. Teniendo en cuenta la velocidad (eje de abcisas) y la precisión (eje de ordenadas) se observa lo potente que es esta red. Aunque se puede ver en la página web de Darknet [Dar, 2018], este es el gráfico:

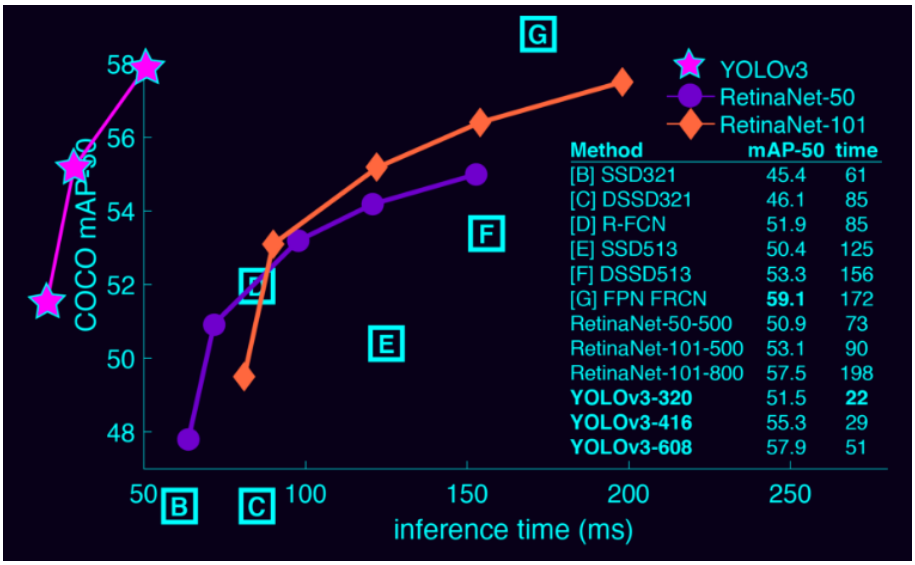


Figura 5.1: Comparación de YOLO con otros sistemas de detección de objetos

Funcionamiento

Los sistemas de reconocimiento de objetos anteriores a YOLO reutilizan clasificadores o localizadores para llevar a cabo dicho reconocimiento. Aplican el modelo obtenido a una imagen en múltiples regiones y con distintos tamaños. YOLO, en cambio, lo enfoca de forma totalmente distinta. Aplica una sola Red Neuronal a la imagen entera, de forma que esta red divide la imagen en distintas partes y predice las *bounding boxes* (rectángulos que se usan para marcar cada objeto detectado) y probabilidades para cada una de estas partes. Las probabilidades predecidas sirven para asignar los pesos a las *bounding boxes*.

YOLO tiene algunas ventajas respecto a los sistemas basados en clasificadores explicados anteriormente. Esto se debe a que tiene en cuenta la totalidad de la imagen. Además, es capaz de realizar predicciones con una sola evaluación de la red, mientras que otros sistemas como Fast R-CNN hacen muchas más. Esto hace que YOLO sea bastante más veloz.

Arquitectura

A continuación se mostrará la arquitectura utilizada para llevar a cabo la extracción de características en la nueva **YOLOv3**. Esta arquitectura es una mezcla entre la que se usaba en YOLOv2 y Darknet-19. Esta nueva versión usa capas convolucionales 3 x 3 y 1 x 1 sucesivas, aunque tiene algunas conexiones que usan atajos. La red tiene 53 capas convolucionales que tienen la siguiente estructura:

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figura 5.2: Arquitectura de YOLO

5.1.2. Primeros pasos

Lo primero que se debe hacer para poder usar YOLO es tener instalado Darknet en el equipo correctamente. Dado que se va a utilizar con un modelo previamente entrenado para no tener que pasar por la tediosa fase de entrenamiento, es necesario descargar los pesos. Para ello, hay que estar situado en el directorio "darknet" e introducir el comando correspondiente:

Listado 5.1: Descarga de los pesos de YOLO

```
cd <darknet_path>
wget https://pjreddie.com/media/files/yolov3.weights
```

Detección en una sola imagen

Obtenidos los pesos, se puede hacer una primera prueba con YOLO. Se le va a pasar una imagen a la red para probar que todo esté en orden y que el detector funcione correctamente. Para ello, se debe ejecutar el siguiente comando:

Listado 5.2: Predicción sobre una imagen

```
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

Si todo ha ido bien, debería aparecer un *output* parecido a este:

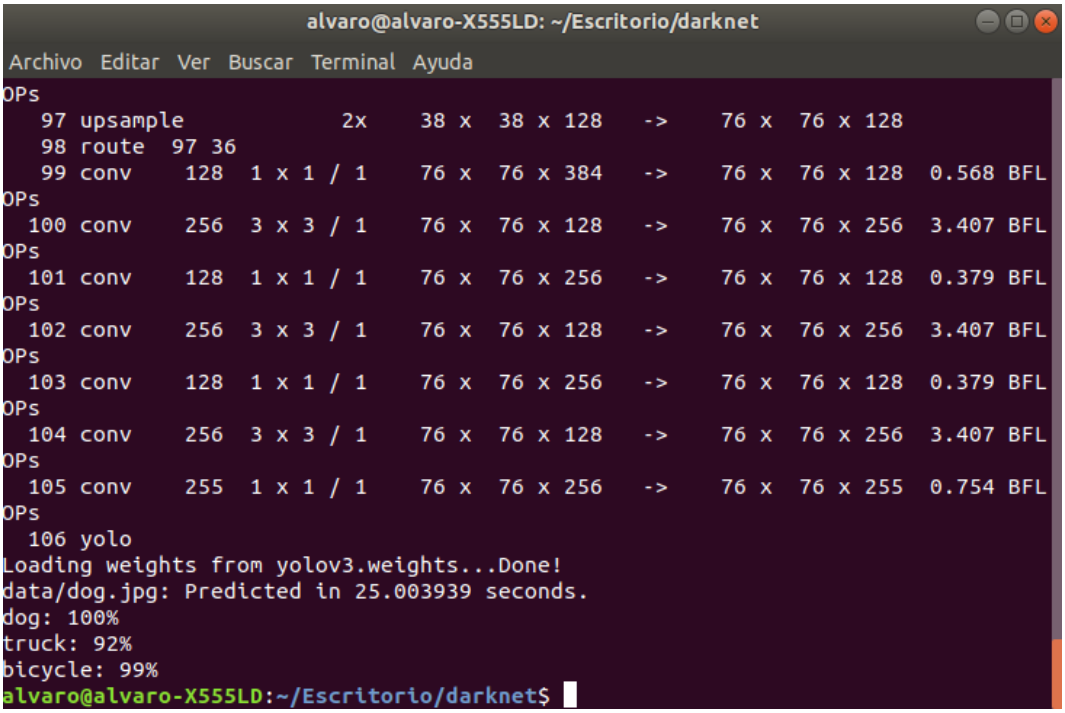


Figura 5.3: Predicción sobre una imagen

Para ver el resultado hay que ir al directorio darknet y abrir un archivo llamado ***predictions.png***:

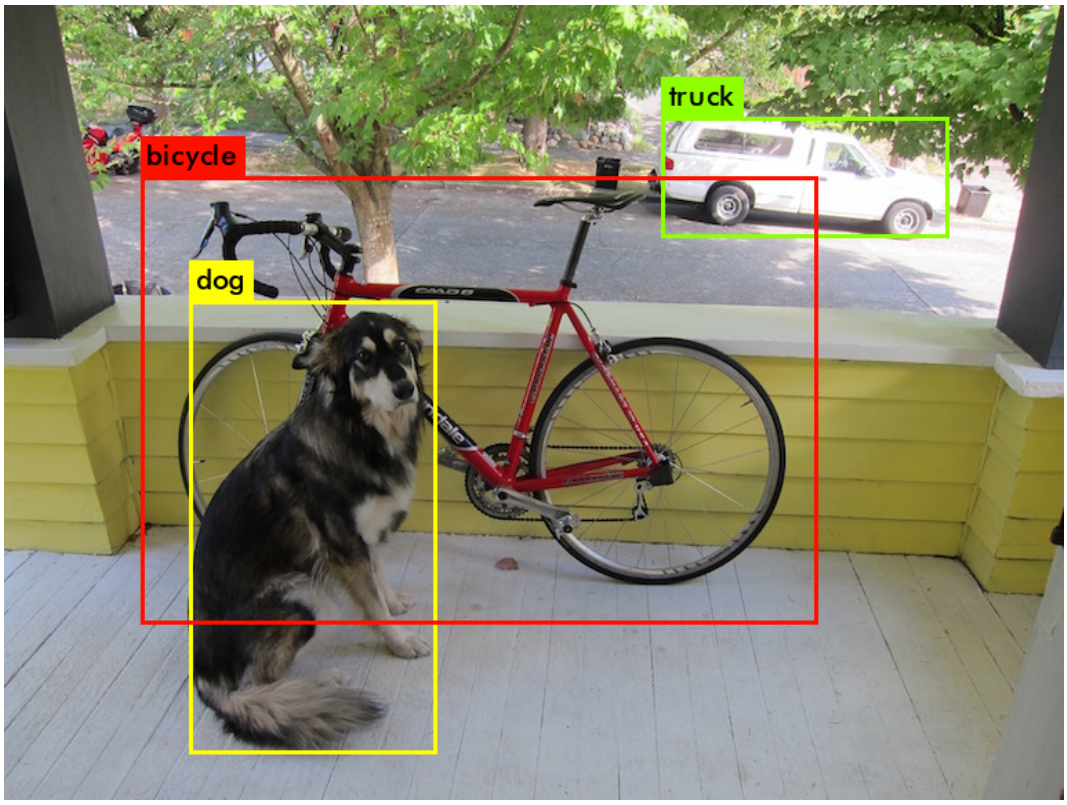


Figura 5.4: Objetos reconocidos por YOLO

Como se puede observar, YOLO ha detectado tres objetos en esta imagen. Un perro, una bicicleta y una furgoneta. De la misma forma que se han detectado objetos en esta imagen, YOLO puede detectar objetos en muchas otras imágenes, siempre que en estas haya alguno de los objetos para los que ha sido entrenada. Es posible hacer más pruebas de detección con cualquiera de las imágenes del directorio "**darknet/data**".

Detección en múltiples imágenes

También es posible dejar en constante ejecución la red e ir pasándole la imagen que se quiere predecir en cada momento. Para ello, se usa el mismo comando que antes, pero esta vez no se especifica la ruta a ninguna imagen:

Listado 5.3: Detección en múltiples imágenes

```
./darknet detect cfg/yolov3.cfg yolov3.weights
```

Una vez cargada la red y los pesos, el sistema pedirá una imagen:

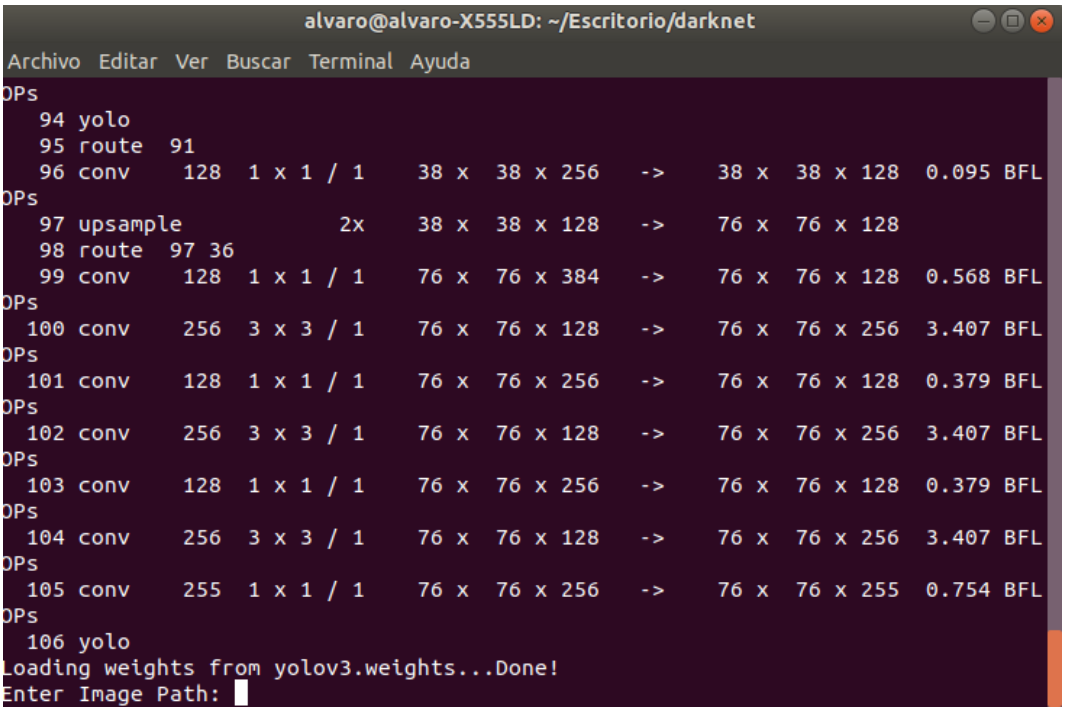


Figura 5.5: Detección en múltiples imágenes

Si se introduce una ruta válida, el sistema procesará la imagen y a continuación pedirá otra. Este proceso se repetirá hasta que se pare la ejecución pulsando **Ctrl-C**

Cambio del límite de detección

Por defecto, YOLO solo muestra objetos detectados con una precisión igual o mayor a 0.25. Esto es lo que se llama límite de detección y puede ser cambiado modificando ligeramente el comando. Para establecer un valor como límite hay que añadir el *flag* **-tresh <val>**, donde <***val***> es el valor deseado. Por tanto, si se quisiera establecer un límite de 0.5 el comando sería:

Listado 5.4: Predicción cambiando el límite de detección

```
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg -thresh 0.5
```

5.1.3. Tiny YOLOv3

Tiny YOLOv3 es un modelo de YOLO muy pequeño diseñado para situaciones especiales en las que no se dispone de muchos recursos. Se usa de la misma forma que YOLO, lo único que hay que hacer es descargar sus pesos. Para ello:

Listado 5.5: Descarga de los pesos de Tiny YOLO

```
wget https://pjreddie.com/media/files/yolov3-tiny.weights
```

Hecho esto, se podría hacer la misma predicción que antes usando el mismo comando, pero habría que cambiar la ruta de los pesos y la configuración. El comando quedaría así:

Listado 5.6: Predicción con Tiny YOLO

```
./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights data/dog.jpg
```

Dado que el modelo es distinto, es posible que cambien los objetos que se detectan.

5.1.4. YOLO en webcam

Esta es una de las características más interesantes de YOLO. Consiste en conectar una webcam al ordenador y detectar los objetos que se tienen alrededor en tiempo real. Para poder usar YOLO en webcam se debe haber realizado la instalación avanzada explicada anteriormente. Además, hay que asegurarse de que la webcam esté conectada y funcionando. Una vez comprobado, se puede ejecutar fácilmente con el comando:

Listado 5.7: Predicción en webcam

```
./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights
```

En la siguiente imagen se puede observar cómo se ejecuta de forma correcta, detectándome a mí como "***person***" (persona), incluso en el espejo que tengo a mi espalda:

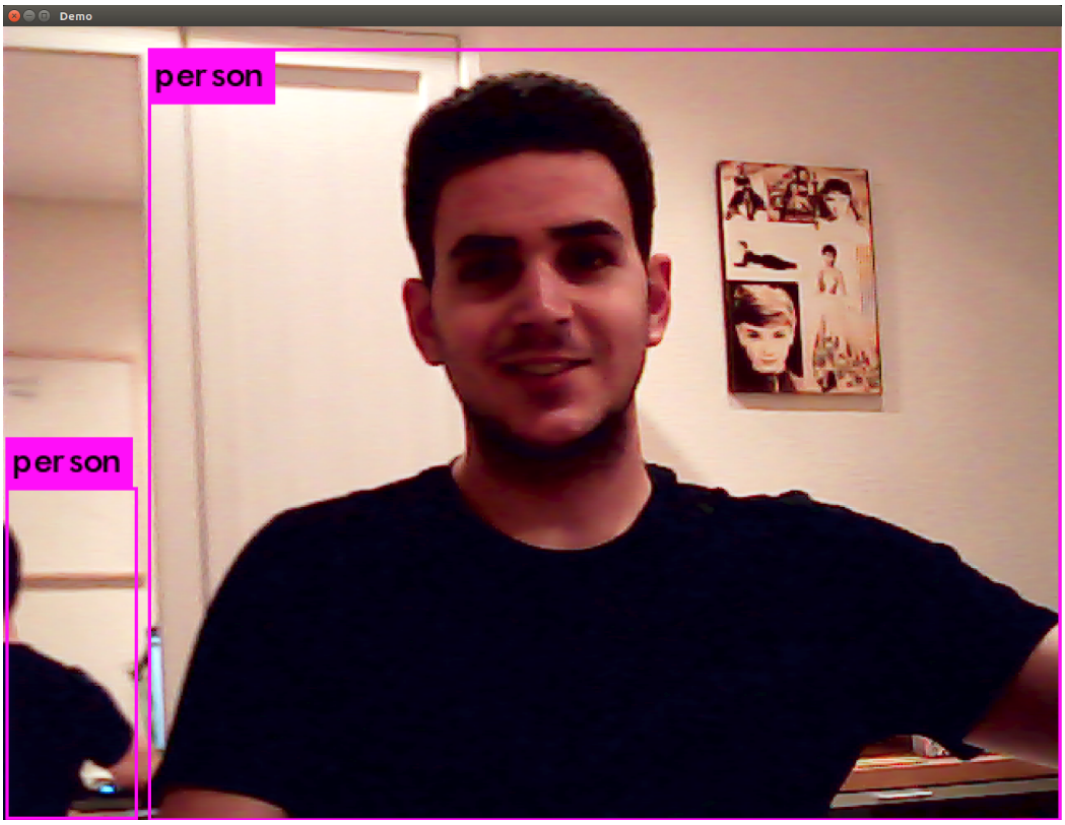


Figura 5.6: Demostración de YOLO en webcam

5.1.5. YOLO en vídeo

De la misma forma que se ha utilizado YOLO con la webcam, se puede utilizar para detectar objetos en vídeos. Esto se hace con el mismo comando, la única diferencia es que ahora hay que espeficiar la ruta al archivo de vídeo.

Listado 5.8: Comando para predicciones en archivos de vídeo

```
./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights <video file>
```

Yo, personalmente, he hecho una prueba en un vídeo descargado de internet. He cargado el vídeo de la siguiente forma:

Listado 5.9: Predicción en un archivo de vídeo

```
./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights ~/Descargas/
    animales_domesticos.mp4
```

Este es un simple vídeo en el que aparecen animales domésticos jugando entre ellos. Aquí se puede ver una captura de pantalla de un momento concreto en el que YOLO

estaba realizando predicciones sobre el vídeo:

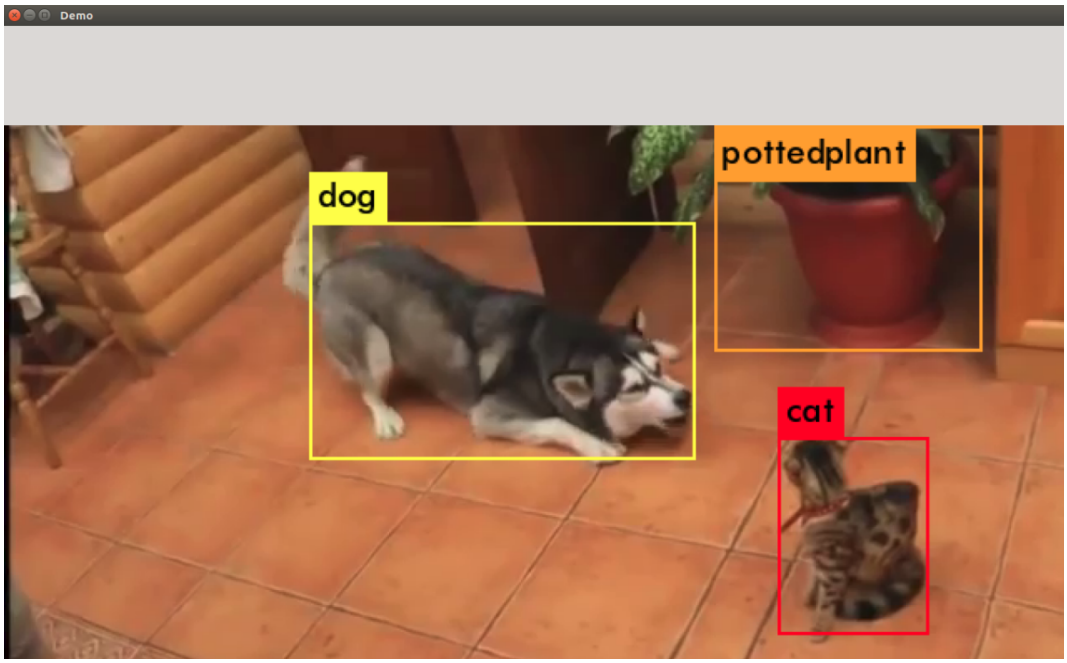


Figura 5.7: Objetos detectados en un archivo de vídeo

5.2. Aportaciones

5.2.1. Integración de YOLO con ROS

Este punto se puede considerar como el principio del proyecto en cuestión, ya que todo lo realizado hasta ahora ha consistido en aprendizaje, configuraciones y pruebas con las herramientas que se utilizarán. Aquí se puede ver un pequeño cuadro-resumen de todo lo que se debería tener llegado este momento:

HERRAMIENTA	ESTADO	VERSIÓN
ROS	Instalado	Kinetic
Nvidia drivers	Instalado	390.48
CUDA Toolkit	Instalado	9.0
OpenCV	Instalado	3.4.0
Darknet	Compilado con GPU y OpenCV	Más reciente
YOLO	Funcionando en todo tipo de imágenes	3

Tabla 5.1: Resumen de herramientas necesarias

Las versiones de estas herramientas son las compatibles con mi equipo y con mi sistema operativo, no se asegura que esta misma instalación funcione en cualquier otro equipo.

Nodo de detección de objetos

La forma mediante la cual se ha integrado YOLO con ROS ha sido por medio de un nodo. Este nodo se encarga de las siguientes tareas:

- Obtener las imágenes de la cámara del robot.
- Cargar la red YOLO para poder trabajar con ella.
- Usar YOLO para procesar cada una de las imágenes obtenidas y detectar los objetos que haya en ellas.

Antes de nada cabe destacar, que esta primera versión del código fue implementada para que funcionase en un mundo virtual simulado con **Gazebo**. Esta no es la versión compatible con Pepper, la cual se mostrará más adelante.

El código

A continuación, se explica paso a paso como se ha implementado este nodo y cada una de sus funcionalidades:

1. Lista de librerías que se deben importar:

Listado 5.10: Librerías importadas

```
1  #!/usr/bin/env python
2  import rospy
3  import cv2
4  import roslib
5  import sys
6  import socket
7  import os
8  from cv_bridge import CvBridge, CvBridgeError
9  from sensor_msgs.msg import Image
10 from tuple_msg.msg import Tuple
11
12 sys.path.append(os.path.join(os.getcwd(), '/home/alvaro/Escritorio/tfg/
       darknet/python/'))
13
14 import darknet as dn
15 import pdb
```

Como se puede observar, antes de importar la propia librería de Darknet, hay que decirle al sistema donde buscarla. En mi caso la ruta correcta es:

"**/home/alvaro/Escritorio/tfg/darknet/python/**"

2. Punto de partida del nodo:

Listado 5.11: Inicio del nodo

```
def main():
        # Set Darknet params
        dn.set_gpu(0)
        node = YoloPub()
        try:
                node.run()
        except rospy.ROSInterruptException:
                pass


if __name__ == '__main__':
        main()
```

Se define el método **main**, que es el primer método en el flujo de ejecución. En este método se establece la GPU que se usará, en mi caso la número 0, ya que es la única existente en mi equipo. Además, se crea una instancia de la clase **YoloPub**, que es la que encapsula todas las funcionalidades del nodo. Por último, se llama al método **run** de esta clase, que se ejecutará hasta que el usuario decida interrumpirlo por teclado (Ctrl + C).

3. Clase principal del nodo e inicializador:

Listado 5.12: Método inicializador de la clase principal

```
class YoloPub:

        def __init__(self):
                rospy.init_node('YoloPub', anonymous=True)
                self.bridge = CvBridge()
                self.image_sub = rospy.Subscriber("camera/rgb/image_raw",
                    Image, self.imageCallback)
                self.imageToProcess = None
                self.pub = rospy.Publisher('yolo_topic', Tuple, queue_size
                    =10)
                self.rate = rospy.Rate(10)

                cfgPath =  "/home/alvaro/Escritorio/tfg/darknet/cfg/yolov3.
                    cfg"
                weightsPath = "/home/alvaro/Escritorio/tfg/darknet/yolov3.
                    weights"
                dataPath = "/home/alvaro/Escritorio/tfg/darknet/cfg/coco2.
                    data"
                self.net = dn.load_net(cfgPath, weightsPath, 0)
                self.meta = dn.load_meta(dataPath)
                self.fileName = 'predict.jpg'
```

Aquí se crea la clase mencionada anteriormente, **YoloPub**, con su método de inicialización, **__init__**, donde se hace lo siguiente:

- Inicialización del nodo mediante el método **init_node** de **rospy**, dándole el

mismo nombre que a la clase y estableciendo el argumento **anonymous** a **True** para especificar que no se tienen en cuenta los nombres duplicados.

- Creación de una instancia de **CvBridge**, que sirve para transformar los mensajes con imágenes de ROS a imágenes de OpenCV.
- Suscripción al tópico "**camera/rgb/image_raw**", del cual se obtienen las imágenes del robot simulado por Gazebo. Las imágenes se obtienen mediante un método callback que será explicado a continuación.
- Creación de la variable **imageToProcess**, que almacenará la imagen sobre la que se haga la predicción.
- Creación de un tópico llamado **yolo_topic** cuya función es publicar los datos que se obtengan con YOLO. Este tópico publica mensajes de tipo **Tuple**, que es un tipo de mensaje creado por mí a través de un proceso que se explicará en otra sección. El argumento **queue_size** sirve para limitar la cantidad de mensajes que hay en la cola si ningún suscriptor los está recibiendo lo suficientemente rápido.
- Creación de un objeto **Rate** que servirá para determinar la velocidad a la que se ejecuta el bucle principal. Puesto que el parámetro que le he pasado ha sido un 10, el bucle se ejecutará 10 veces por segundo.
- Creación de las variables **cfgPath**, **weightsPath** y **dataPath** que contienen la ruta al fichero de configuración, de pesos y de datos respectivamente. La configuración y los pesos elegidos son, lógicamente, los de la versión 3 de YOLO. El archivo de datos es el de COCO, pero ligeramente modificado para obtener la ruta correcta. Simplemente se ha cambiado esta línea:

Listado 5.13: Ruta incorrecta a la lista de nombres

```
names = data/coco.names
```

Por esta otra:

Listado 5.14: Ruta a la lista de nombres

```
names = /home/alvaro/Escritorio/tfg/darknet/data/coco.names
```

- Declaración de la variable **net**, en la cual se ha cargado la red. Para poder cargar la red se usa el método **load_net** de Darknet, que toma como argumentos los archivos de configuración y pesos.
- Declaración de la variable **meta**, donde se cargan los datos referentes a la red. Para ello se usa la función **load_meta**, que toma como argumento el fichero de datos.
- Creación de una variable llamada **fileName** para almacenar el nombre con el cual las imágenes a predecir serán guardadas en el sistema. El funcionamiento

es el siguiente: se guarda la primera imagen a predecir, se realiza la predicción obteniendo los objetos reconocidos, se sobreescribe la primera imagen con la siguiente y se repite el proceso.

4. Método callback de obtención de imágenes:

Listado 5.15: Función para recibir las imágenes de la cámara

```
def imageCallback(self, data):
        self.imageToProcess = self.bridge.imgmsg_to_cv2(data, "bgr8")
```

Este método utiliza el **bridge** declarado anteriormente para transformar las imágenes de la cámara del robot en imágenes con formato OpenCV. Esta función es un **callback**, lo que significa que es una función a la que no llama el propio usuario, sino que se delega en otro componente para llamarla. Tal y como se ha definido el suscriptor en el método **__init__**, se hace una llamada a esta función cada vez que se recibe una nueva imagen de la cámara.

5. Método que implementa el bucle principal del programa:

Listado 5.16: Método principal del nodo

```
def run(self):

        while not rospy.core.is_shutdown():

                if(self.imageToProcess is not None):
                        cv2.imwrite(self.fileName, self.imageToProcess)
                        r = dn.detect(self.net, self.meta, self.fileName)
                        print r

                        # Get number of objects detected
                        num_objects = len(r)

                        # Coordenates index
                        x_index = 0
                        y_index = 1
                        width_index = 2
                        height_index = 3

                        # Lists for objects data
                        names = []
                        x = []
                        y = []
                        widths = []
                        heights = []

                        # Loop for managing data
                        for obj in range(0, num_objects):
                                names.append(r[obj][0])
                                x.append(float(r[obj][2][x_index]))
                                y.append(float(r[obj][2][y_index]))
                                widths.append(float(r[obj][2][width_index]))
                                heights.append(float(r[obj][2][height_index
                                    ]))

```

```
34                          data = Tuple()
35                          data.names = names
36                          data.x = x
37                          data.y = y
38                          data.widths = widths
39                          data.heights = heights
40                          self.pub.publish(data)
41                          self.rate.sleep()
```

Se define el método **run**, el cual implementa un bucle que se ejecuta hasta que es detenido por el usuario, mediante la instrucción **while not rospy.core.is_shutdown()**. Dentro de este bucle siempre se comprueba que la variable que contiene la imagen no tenga valor nulo. En caso de que esto se cumpla, se realizan las siguientes acciones:

- Almacenamiento de la imagen en el equipo mediante la función **imwrite** de **OpenCV**. En caso de que haya una guardada con el mismo nombre (el que esté almacenado en la variable **fileName**), se sobreescribirá.
- Obtención de la lista de objetos reconocidos llamando a la función **detect** de Darknet. Las variables que toma como argumento han sido explicadas anteriormente.
- Creación de la variable **num_objects**, donde se almacena la cantidad de objetos reconocidos. Se usa la función **len** de Python sobre la lista de objetos para obtener dicha cantidad.
- Creación de constantes que almacenan índices para acceder al elemento deseado del vector. La lista que devuelve YOLO tiene la siguiente estructura para cada objeto detectado:
  - nombre
  - precision
  - (posicion x, posicion y, ancho, alto)

  Por lo tanto, cada objeto detectado tiene los siguientes componentes:
  - **Nombre**: cadena de texto.
  - **Precisión con la que se detecta el objeto**: Número decimal.
  - **Datos relativos a la posición del objeto**: Array.

  Las constantes **x_index**, **y_index**, **width_index** y **height_index** se utilizan para acceder a los valores de este Array de datos posicionales.
- Creación de listas para organizar los datos de todos los objetos reconocidos. Cada lista sirve para almacenar un dato concreto, por ejemplo, en la lista **names** se almacenarán los nombres de los objetos.
- Bucle que itera sobre la lista de objetos reconocidos y, para cada objeto, añade cada uno de los datos a la lista correspondiente. Para introducir los datos en las lista se utiliza la funcion **append**.

- Creación de un objeto de tipo **Tuple** y asignación del valor correspondiente a cada una de sus variables. Como ya he dicho antes, el tipo **Tuple** ha sido creado por mí, es por eso que tiene todos los atributos que se necesitan publicar.
- Publicación de la tupla usando el **Publisher** creado anteriormente.

Funcionamiento

Ahora se va a hacer una demostración paso por paso de cómo funciona el nodo descrito anteriormente y de cómo se ha conseguido integrar YOLO en ROS con éxito. Antes de dicha demostración, es necesario asegurarse de tener disponible el simulador a utilizar. En este caso se va a utilizar **Gazebo** para simular un **Turtlebot** en un mundo vacío. Para poder poner en marcha esta simulación, lo primero que hay que hacer es instalar todas las dependencias de **Turtlebot** mediante el comando:

Listado 5.17: Descarga e instalación de todas las dependencias de Turtlebot

```
sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-
    turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-kobuki-ftdi
     ros-kinetic-ar-track-alvar-msgs
```

Cuando termine la instalación, hay que ir al directorio **catkin_ws/src** y descargar el paquete de simulación:

Listado 5.18: Descarga del paquete de simulación

```
git clone https://github.com/turtlebot/turtlebot_simulator
```

Descargado este paquete, se puede comenzar con la demostración:

1. Abrir una nueva terminal e introducir el comando:

Listado 5.19: Inicio de ROS

```
roscore
```

2. En una terminal distinta, ir al directorio **catkin_ws**, compilar y lanzar el simulador.

Listado 5.20: Ejecución del mundo simulado

```
cd catkin_ws
catkin_make
source devel/setup.bash
roslaunch turtlebot_gazebo turtlebot_world.launch
```

3. En otra terminal nueva, ir al directorio **catkin_ws** y lanzar el nodo creado.

Listado 5.21: Ejecución del nodo de YOLO

```
cd catkin_ws
source devel/setup.bash
rosrun tfg yolo_pub.py
```

Si todos estos pasos se llevan a cabo correctamente, el resultado debería ser el siguiente:

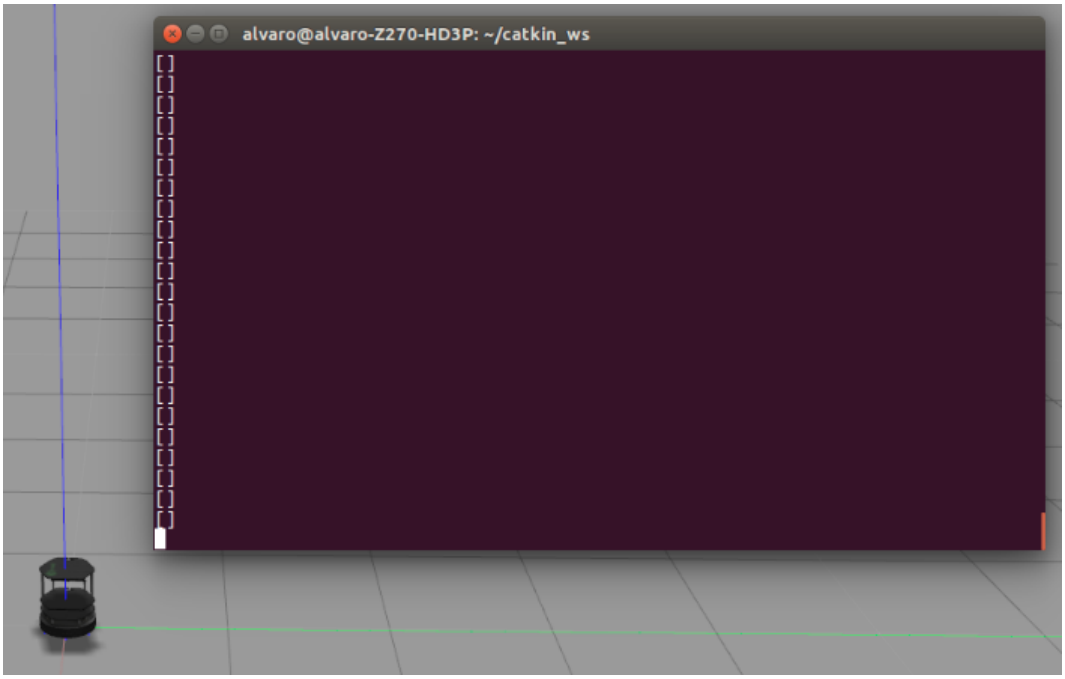


Figura 5.8: Ejecución del nodo de YOLO

Como se puede observar, la lista de objetos reconocidos está vacía. Esto se debe a que en el mundo simulado no hay ningún objeto que YOLO sea capaz de reconocer. En cambio, esto es lo que pasaría si se añadiera, por ejemplo, un bol:

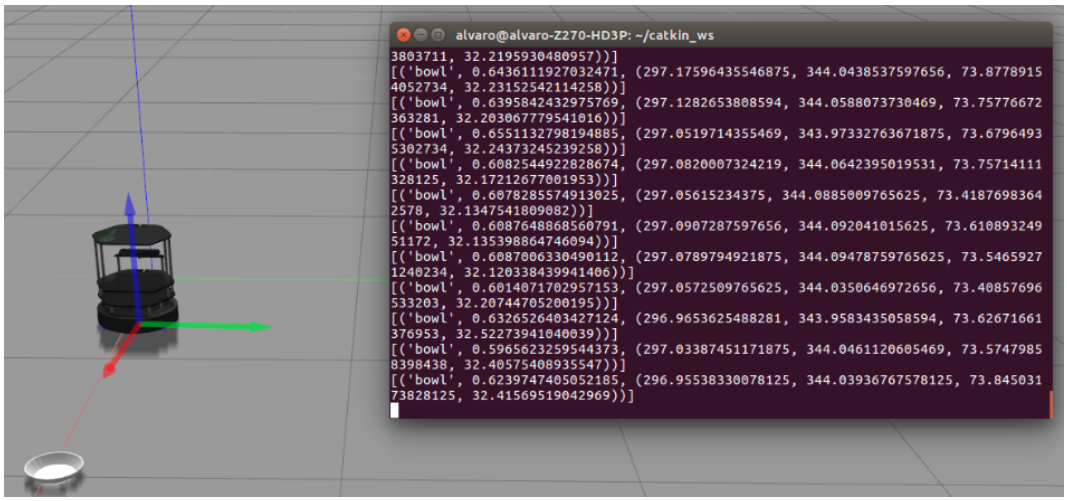


Figura 5.9: Detección de un bol usando el nodo

En este caso, YOLO lo reconoce sin problema.

Visualizador con OpenCV

Ahora que ya está comprobado que el nodo funciona sin ningún problema, el siguiente paso es que no solo reconozca los objetos, sino que también los muestre y los identifique. Para ello se van a usar funciones de **OpenCV** y los datos de la lista de objetos que corresponden con las posiciones.

El código

Lo primero que se debe hacer es declarar la fuente que se va a usar para escribir los nombres de los objetos en la imagen. Yo la he declarado en el método **__init__** de forma que pueda usarla en cualquier parte. Esta es la que voy a usar:

Listado 5.22: Declaración de la fuente de OpenCV para el nombre de los objetos

```
1 self.font = cv2.FONT_HERSHEY_TRIPLEX
```

El resto de código para implementar el visualizador debe estar situado dentro del bucle principal, de forma que se actualice para cada imagen que se reciba. El código es el siguiente:

Listado 5.23: Código que implementa el visualizador

```
1 img = cv2.imread(self.fileName)
2
3 for i in range(0, len(names)):
4         name = names[i]
```

```
5           pt1 = (int(x[i] - (widths[i]/2)), int(y[i] - (heights[i]/2)))
6           pt2 = (int(x[i] + (widths[i]/2)), int(y[i] + (heights[i]/2)))
7           cv2.rectangle(img, pt1, pt2, (0,0,0), thickness=2, lineType=8, shift=0)
8           name_start_point = ((int(x[i] - (widths[i]/2)), int(y[i] - (heights[i]/2)
                - 8)))
9           cv2.putText(img,name,name_start_point, self.font, 0.4, (0,0,0), 1 ,cv2.
                LINE_AA)
10
11  cv2.imshow('image',img)
12  cv2.waitKey(1)
```

Y funciona de la siguiente forma:

1. Se crea una variable que contendrá la imagen en un formato compatible con **OpenCV**

2. Se crea un bucle que iterará sobre todos los objetos detectados. Para cada objeto:

   a) Se obtiene su nombre.

   b) Se obtiene la localización del vértice superior izquierdo del rectángulo que identifica al objeto y se guarda en la variable **pt1**. Esta localización debe ser dada en coordenadas **(x, y)**, ya que es una imagen en 2D. Más adelante se muestra una imagen de los cálculos que se han hecho para obtener los puntos.

   c) Se obtiene la localización del vértice inferior derecho del rectángulo y se almacena en la variable **pt2**.

   d) Se dibuja el rectángulo deseado en la imagen mediante el método **rectangle** de **OpenCV**. Los argumentos de esta función son: la imagen, los puntos calculados anteriormente y otros de propósito estético (grosor de línea, tipo de línea, etc).

   e) Se calcula la posición donde irá el texto. Realmente esta posición va a gusto del usuario. Yo la he puesto un poco más arriba del vertice superior izquierdo del rectángulo.

   f) Se añade el texto a la imagen mediante el método **putText** de **OpenCV**. Los argumentos que recibe esta función son: la imagen, el texto que se desea introducir (en este caso el nombre del objeto), la posición donde se desea que empiece el texto y otros de carácter estético (tipo de fuente, tamaño del texto, etc).

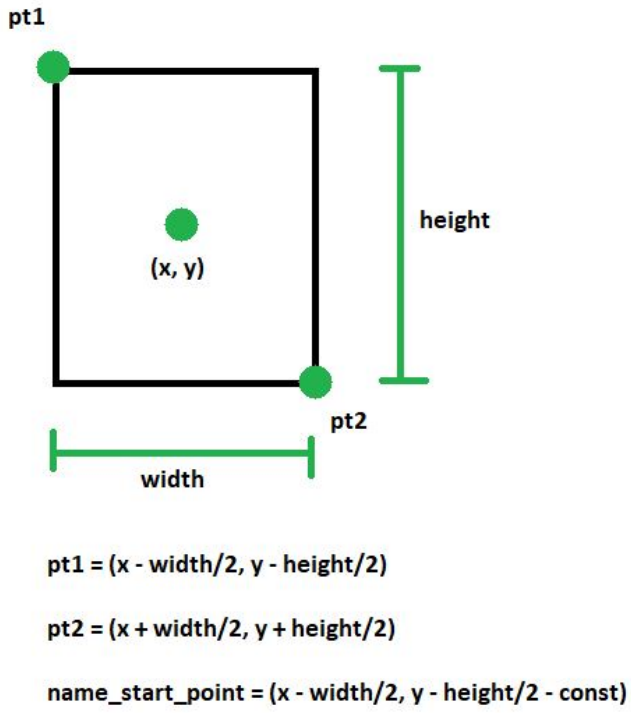


Figura 5.10: Explicación de los cálculos de las coordenadas

3. Se muestra la imagen mediante el método **imshow** de **OpenCV**.

4. Se usa la función **waitKey** de **OpenCV** con 1 como argumento para que se muestre un fotograma por milisegundo.

Funcionamiento

Dado que esta nueva funcionalidad se ha añadido sobre el nodo creado anteriormente, lo único que hay que hacer para ejecutarlo es seguir los mismos pasos.

El resultado de esta nueva funcionalidad sobre el ejemplo del bol, mostrado anteriormente, sería este:

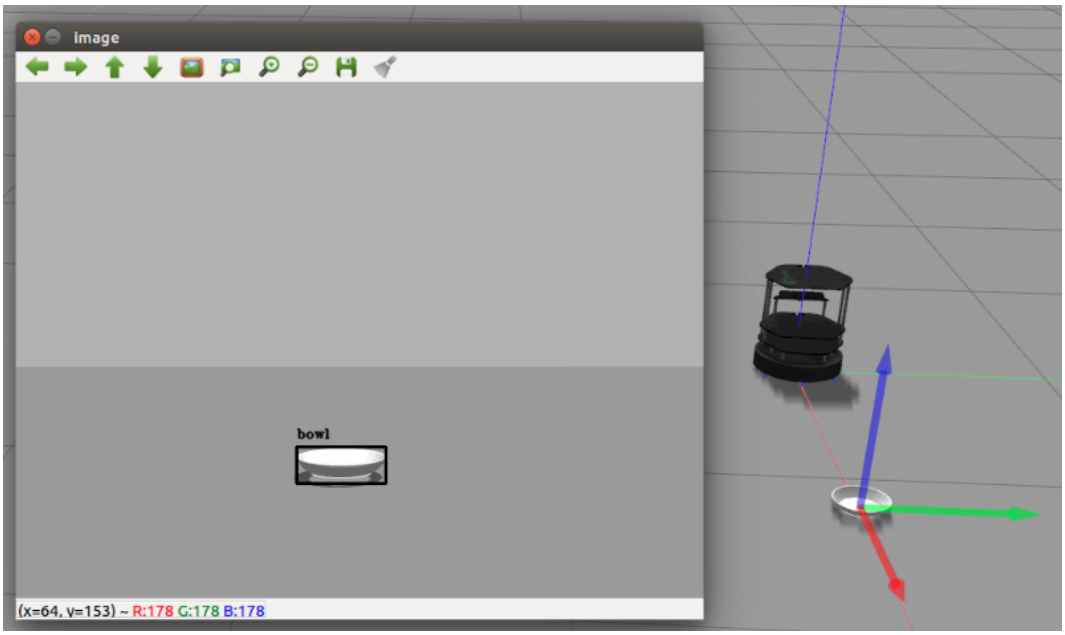


Figura 5.11: Identificación de un bol usando el nodo de YOLO con OpenCV

Mensaje para publicar los datos

Como se ha visto en apartados anteriores, mediante la integración de YOLO con ROS se ha conseguido reconocer objetos usando un robot simulado. El hecho de haber reconocido los objetos está muy bien, pero para darle a esto un uso práctico hay que ser capaz de publicarlos con un tópico, de modo que haya otros nodos que puedan suscribirse a este tópico, obtener los datos y usarlos con algún propósito.

El problema de esto, es que ROS no integra todo tipo de datos en la publicación de sus mensajes como, por ejemplo, las listas. Para poder publicar esta clase de datos hay que crear un mensaje personalizado. En mi caso, he creado un tipo de mensaje llamado **Tuple** que me ha permitido publicar los datos que necesitaba.

Antes de explicar cómo crear un mensaje personalizado, es importante destacar que ROS sí que integra algunos mensajes para poder publicar tipos de datos simples como Booleanos, Enteros o Strings entre otros. Estos mensajes están recogidos en una librería llamada **std_msgs**.

Para crear un mensaje personalizado se deben seguir los siguientes pasos:

1. Hay que crear un nuevo paquete en el directorio **catkin_ws**, que contendrá la implementación y configuración del mensaje. En mi caso, el paquete se llama **tuple_msg**, por lo que el comando utilizado ha sido:

Listado 5.24: Creación del paquete que contendrá el mensaje

```
catkin_create_pkg tuple_msg
```

2. Cuando el paquete esté creado, se debe ir al nuevo directorio y crear una carpeta que contenga el archivo que implementa el mensaje. Yo he llamado a esta carpeta **msg** y al archivo **Tuple.msg**. Todo esto se resume en los siguientes comandos:

Listado 5.25: Creación del archivo que implementa el mensaje

```
cd catkin_ws/src/tuple_msg
mkdir msg
touch Tuple.msg
```

3. Este nuevo paso consiste en la implementación del mensaje con los tipos de datos que se deseen. En este caso concreto, se necesitan publicar los siguientes datos:

   - **Names**: Array de Strings que contendrá cada uno de los nombres de los objetos reconocidos
   - **x**: Array de números decimales que contendrá la coordenada x de cada uno de los objetos.
   - **y**: Igual que el array anterior, pero este contendrá la coordenada y de cada objeto.
   - **widths**: Array de números decimales que contendrá la anchura de cada objeto reconocido.
   - **heights**: Este array será igual que el anterior, pero en vez de la anchura contendrá la altura de los objetos.

   Por tanto, la implementación final del mensaje quedaría así:

Listado 5.26: Implementación del mensaje

```
1 string[] names
2 float32[] x
3 float32[] y
4 float32[] widths
5 float32[] heights
```

4. Ahora hay que modificar algunas líneas del archivo **CMakeLists.txt**, de modo que el mensaje esté bien configurado al compilar. Lo primero que hay que cambiar es la línea:

Listado 5.27: Dependencias del paquete

```
find_package(catkin REQUIRED)
```

Por:

Listado 5.28: Adición de componentes necesarios al paquete

```
find_package(catkin REQUIRED COMPONENTS
        message_generation
        std_msgs
)
```

También hay que descomentar las líneas:

Listado 5.29: Configuración inicial de mensajes

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

Y cambiarlas usando el nombre del archivo **.msg** que haya sido creado en los pasos anteriores:

Listado 5.30: Nueva configuración del mensaje

```
add_message_files(
   FILES
   Tuple.msg
)
```

Hecho esto, hay que descomentar las líneas:

Listado 5.31: Activación de dependencias de mensajes

```
# generate_messages(
#    DEPENDENCIES
#    std_msgs
# )
```

Para terminar con la configuración del archivo **CMakeLists.txt**, hay que cambiar estas líneas:

Listado 5.32: Configuración inicial de dependencias Catkin

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES tuple_msg
#  CATKIN_DEPENDS other_catkin_pkg
#  DEPENDS system_lib
)
```

Por estas otras:

Listado 5.33: Nueva configuración de dependencias Catkin

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES tuple_msg
  CATKIN_DEPENDS message_runtime
#  DEPENDS system_lib
)
```

5. Por último, hay que configurar el archivo **package.xml** añadiéndole unas líneas. Debajo de la línea de **buildtool_depend** hay que añadir estas otras:

Listado 5.34: Configuración del archivo que identifica el paquete

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Una vez hecho todo esto, y tras la correspondiente compilación del directorio catkin, ya se podría usar el mensaje creado para publicar datos.

5.2.2. Traslado a Pepper

Después de todo este proceso y tras haber comprobado que YOLO funciona perfectamente en el simulador Gazebo, es hora de intentar cumplir el objetivo de este proyecto y probarlo en el robot Pepper. Para que no haga falta tener al Pepper delante cada vez que se quiere hacer una prueba, se ha utilizado lo que se llama un **rosbag** para hacer las primeras pruebas con este robot. A continuación se explica en que consiste esta función de ROS.

Rosbag

Un **rosbag** es un tipo de archivo de ROS que encapsula todos los tópicos que había activos durante la grabación del mismo, de modo que puedan usarse en cualquier otro momento. Así que se grabó un archivo de este tipo en el laboratorio para poder trabajar con el Pepper sin tenerlo físicamente. Se preparó una escena en la que el Pepper estaba frente a una serie de objetos que YOLO es capaz de reconocer, de esta forma he podido probar el nodo anterior en este robot sin ningún tipo de problema.

Además, este tipo de archivo permite la reproducción en bucle del mismo, lo permite utilizarlo de forma continua sin tener que parar a reanudarlo. Para reproducir el archivo en bucle hay que utilizar el comando:

Listado 5.35: Comando de ejecución en bucle del archivo Rosbag

```
rosbag play -l <path_to_bag_file>
```

Para comprobar que se reproduce de forma correcta, lo que se puede hacer es abrir una nueva terminal y escribir:

Listado 5.36: Visualización de los tópicos activos

```
rostopic list
```

Si todo va bien deberían aparecer tópicos del Pepper, tal y como se observa en esta imagen:

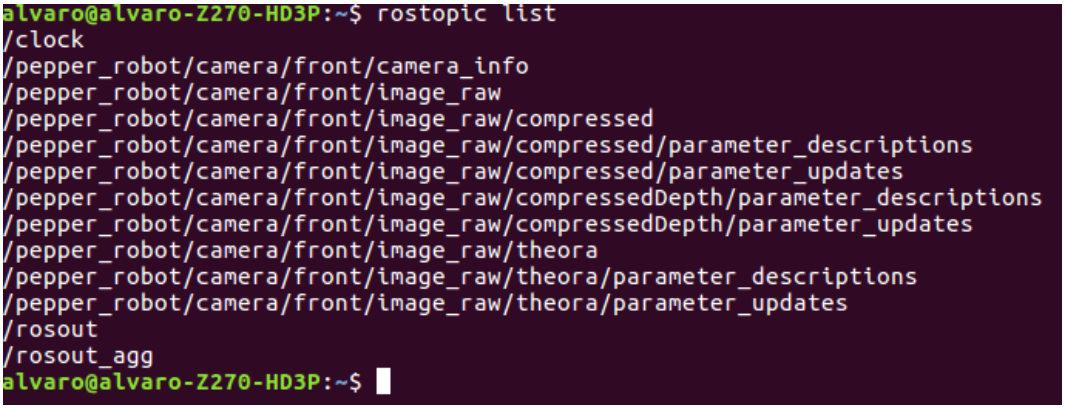


Figura 5.12: Listado de tópicos disponibles

Además, así es posible conocer el nombre del tópico al que hay que suscribirse para obtener las imágenes. En este caso: **/pepper_robot/camera/front/image_raw**. Una vez que se conoce este dato, lo único que hay que hacer para probar el nodo usando el **rosbag** es cambiar el suscriptor. Se debería cambiar esta línea:

Listado 5.37: Inicio del nodo

```
1 self.image_sub = rospy.Subscriber("camera/rgb/image_raw", Image, self.
    imageCallback)
```

Por esta otra:

Listado 5.38: Inicio del nodo

```
1 self.image_sub = rospy.Subscriber("/pepper_robot/camera/front/image_raw", Image,
    self.imageCallback)
```

Funcionamiento

Estos son los pasos a seguir para ejecutar el nodo usando el **rosbag**:

1. Abrir una terminal e iniciar ROS mediante el comando:

Listado 5.39: Inicio de ROS

```
roscore
```

2. Abrir otra terminal y reproducir el rosbag en bucle, en mi caso:

Listado 5.40: Ejecución en bucle del archivo Rosbag

```
rosbag play -l /Descargas/2018-07-20-11-17-30.bag
```

3. En una nueva terminal, ir al directorio **catkin_ws** y ejecutar el nodo:

Listado 5.41: Ejecución del nodo de YOLO

```
cd catkin_ws
source devel/setup.bash
rosrun tfg yolo_pub.py
```

La siguiente imagen es una captura de pantalla tomada durante la ejecución:

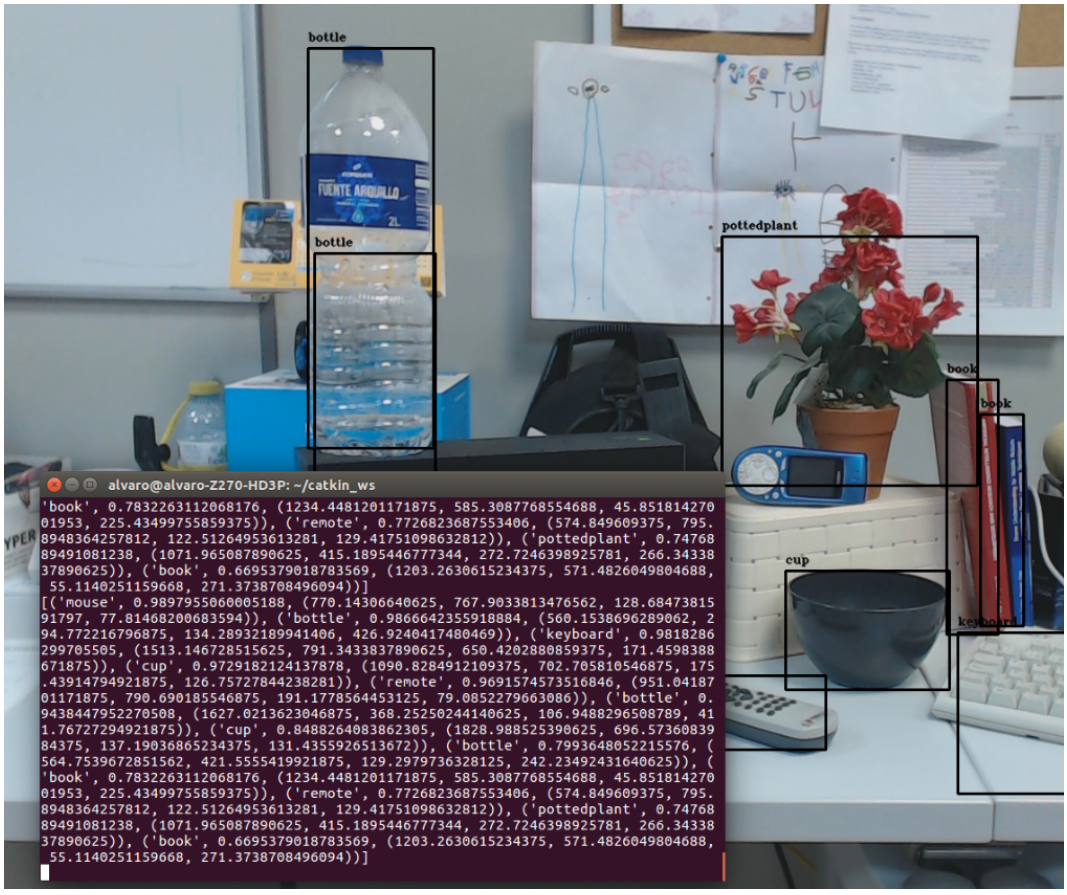


Figura 5.13: Ejecución del nodo de YOLO usando el rosbag

La imagen está cortada, pero como se aprecia, se reconocen los objetos de la cámara del Pepper y el visualizador funciona perfectamente.

Pepper Real

Aunque el rosbag es muy útil para hacer pruebas, hay que recordar que el proyecto ha sido realizado para que las descripciones las dé un Pepper real. Por eso, se ha hecho funcionar toda la implementación explicada anteriormente en los equipos del laboratorio y se ha probado con el robot original. Del mismo modo que con el rosbag, los resultados han sido satisfactorios. En la sección de experimentación se pueden ver imágenes de estos resultados.

El hecho de haber utilizado ROS para realizar este proyecto ha facilitado mucho la adaptación del mismo a los equipos del laboratorio, ya que este framework permite que la implementación realizada sea compatible con cualquier robot, sensor o simulador. Lo único que hubo que hacer fue copiar el paquete que contiene los nodos en el *Workspace* de *Catkin* de estos equipos y cambiar las rutas mediante las que se obtienen la configuración y datos de Darknet.

La arquitectura de despliegue que se ha utilizado en el laboratorio para poner en marcha la aplicación es la siguiente:

- Robot Pepper mediante la cámara del cual se obtienen las imágenes en las que se detectan los objetos. Es este robot el que se utiliza para reproducir las frases finales que se obtienen del descriptor de escenas.
- Servidor de *Deep Learning* que se utiliza para ejecutar el nodo de YOLO debido a su gran capacidad de cómputo. En este servidor se utiliza una GPU Nvidia Quadro P6000 de 24 *Gigabyte* (GB). Se comunica con el robot Pepper para obtener las imágenes.
- Equipo sencillo que se comunica con el servidor de *Deep Learning* para conseguir los datos de los objetos reconocidos y, a continuación, obtener la descripción de la escena. También se comunica con el robot Pepper para enviarle esta descripción en formato de *speech*.

5.2.3. Descriptor de escenas

Todo el trabajo realizado hasta ahora ha sido bueno y se ha conseguido que un robot sea capaz de reconocer objetos. Pero, siendo realistas, el reconocimiento de objetos de por sí no aporta ningún valor. Es decir, ¿De qué sirve que un robot pueda reconocer objetos si no es capaz de darle un uso práctico?

Por esta razón se ha decidido hacer una aplicación para dar sentido y poner fin a este proyecto. Esta aplicación, como se ha comentado anteriormente, consiste en un descriptor de escenas. Es decir, que el robot no solo sea capaz de reconocer objetos, sino de

usar este reconocimiento para describir lo que tiene delante.

La aplicación se suscribe al tópico donde están publicados los datos de los objetos reconocidos, obtiene estos datos, y trabaja con ellos para sacar conclusiones. A continuación se explica la implementación.

El código

Este es es el punto de partida del nodo:

Listado 5.42: Punto de partida del nodo descriptor de escenas

```
def main():
        node = SceneDescriptor()
        node.run()


if __name__ == '__main__':
        main()
```

Como se observa, en el método principal lo único que se hace es instanciar la clase que contiene toda la implementación (**SceneDescriptor**) y llamar al método principal de esta clase (**run**). A continuación, se crea la clase con su método inicializador, **__init__**:

Listado 5.43: Inicialización de la clase principal del nodo

```
class SceneDescriptor:

        def __init__(self):
                rospy.init_node('SceneDescriptor', anonymous=True)
                rospy.Subscriber("yolo_topic", Tuple, self.callback)
```

En el **__init__** se inicializa el nodo llamándolo **SceneDescriptor** y se realiza la suscripción al tópico **yolo_topic** para obtener los datos extraídos por YOLO. El suscriptor funciona con un método callback, como los que ya se han explicado anteriormente. Este es dicho método:

Listado 5.44: Método callback para obtener los datos de YOLO

```
def callback(self, data):
        self.yolo_data = data
```

Este método solo tiene la función de hacer que la variable **yolo_data**, que es de tipo **Tuple**, almacene los datos que se reciben. Esta es la declaración de dicha variable:

Listado 5.45: Declaración de una varaible del tipo personalizado Tuple

```
yolo_data = Tuple()
```

Después de la inicialización comienza a ejecutarse la función **run**, cuyo código es el siguiente:

Listado 5.46: Función principal del nodo

```
def run(self):

        while not self.yolo_data.names:
                if self.yolo_data.names:
                        main_object = self.middle_object()
                        left = self.left_objects()
                        right = self.right_objects()

                        say_left = self.list_to_string(left)
                        say_right = self.list_to_string(right)

                        print("")
                        print("SCENE DESCRIPTION")
                        print("I can see a " + main_object)
                        print("At the left side of the " + main_object + say_left)
                        print("At the right side of the " + main_object +
                            say_right)
                        print("")
```

Como se puede observar todo el código de la función esta contenido en un bucle while, que itera hasta que la variable contenga los datos publicados por el nodo de YOLO. Si el publicador no estuviese activo, estos datos no se recibirían y el bucle iteraría continuamente.

En cada iteración del bucle se comprueba si la variable **yolo_data** está o no vacía mediante una instrucción **if**. En caso de que esté vacía, no se realiza ninguna acción. Será solo en la última iteración cuando la condición del **if** se cumpla y se trabaje con los datos para obtener la descripción de la escena. La descripción de la escena se obtiene de la forma siguiente:

1. Se obtiene el objeto que el robot ve más centrado llamando al método **middle_object**, cuya implementación es la siguiente:

Listado 5.47: Método que obtiene el objeto más centrado

```
def middle_object(self):

        sorted_list = sorted(self.yolo_data.x)

        middle = float(len(sorted_list))/2
        if middle %2 != 0:
                middle = middle - 0.5

        self.mid_val = sorted_list[int(middle)]

        index = self.yolo_data.x.index(self.mid_val)

        return self.yolo_data.names[index]
```

   En esta función se siguen una serie de pasos para obtener el elemento más centrado (en el eje horizontal):

   a) Se ordena la lista de coordenadas **x** de los objetos de menor a mayor y se guarda en la variable **sorted_list**. Esto se hace de forma sencilla usando

el método **sorted**. Supongamos una lista ficticia para utilizarla a modo de ejemplo:

2	5	1	8	9	3

Así quedaría la lista ordenada:

1	2	3	5	8	9

b) Se obtiene el índice del elemento que está en medio diviendo la longitud de la lista entre 2. Si el resto de esta división es distinto a 0, significa que la lista tiene una cantidad impar de elementos, y hay que restar 0.5 al resultado para que el índice sea correcto. Si el resto es par, quiere decir que la lista tiene una cantidad par de elementos. Este índice se almacena en la variable **middle**. Para seguir con el ejemplo:

$$6 \;/\; 2 = 3$$

c) El siguiente paso es obtener el elemento que está en el índice que se acaba de calcular y guardarlo en la variable **mid_val**. Para ello simplemente hay que acceder a la lista.

$$\text{lista\_ordenada}[3] = 5$$

d) A continuación, hay que obtener el índice de ese mismo elemento en la lista original y almacenarlo en la variable **index**. Esto se puede conseguir con el método **index**.

$$\text{lista\_original.index}(5) = 1$$

e) Finalmente, se accede a la lista de nombres con el índice recién calculado para devolver el objeto más centrado.

return lista_nombres[1]

2. Se obtiene la lista de objetos que hay a la izquierda del objeto central llamando a la función **left_objects**, que está implementada así:

Listado 5.48: Método para obtener la lista de objetos de la izquierda

```
1  def left_objects(self):
2
3          left_objs = []
4
5          for i in range(0, len(self.yolo_data.x)):
6                  if self.yolo_data.x[i] < self.mid_val:
7                          left_objs.append(self.yolo_data.names[i])
8
9          return left_objs
```

Esta implementación es muy sencilla. Lo único que se hace es crear una lista vacía a la que se irán añadiendo los objetos cuyo valor de la **x** sea menor que el del objeto

central, calculado anteriormente. Para llevar esto a cabo, se usa un bucle que itera sobre la lista donde se almacena el valor x de todos los elementos. Finalmente se devuelve la lista.

3. Se obtiene la lista de objetos situados a la derecha del central mediante la función **right_objects**:

Listado 5.49: Método para obtener la lista de objetos de la derecha

```
def right_objects(self):

    right_objs = []

        for i in range(0, len(self.yolo_data.x)):
                if self.yolo_data.x[i] > self.mid_val:
                        right_objs.append(self.yolo_data.names[i])

        return right_objs
```

El funcionamiento de este método es el mismo que el de **left_objects**, con la única diferencia de que ahora se añaden a la lista los objetos cuyo valor de **x** sea mayor que el del objeto central.

4. El siguiente paso, después de obtener las listas, es convertir dichas listas en una cadena de texto que el robot sea capaz de decir. Para ello se ha implementado la función **list_to_string**:

Listado 5.50: Función para convertir las listas en cadenas de texto

```
def list_to_string(self, lst):

        to_say = ""

        if len(lst) == 1:
                to_say += " there is " + self.get_number_objects(lst[0], lst
                    ) + lst[0]
        else:
                to_say += random.choice(self.possible_sentences)

                already_said = []
                for obj in lst:
                        if obj not in already_said:
                                already_said.append(obj)
                                number = self.get_number_objects(obj, lst)
                                if number == "a " or number == "an ":
                                        to_say += number + obj + ", "
                                else:
                                        to_say += number + obj + "s, "

        return to_say
```

Estos son los pasos que sigue la implementación:

a) Se declara un string vacío (**to_say**) que será el que contenga el texto final.

b) Se comprueba el tamaño de la lista que se quiere convertir en texto. En caso de que la lista solo tenga un elemento, el string comenzará con la partícula "**there is**" (hay, en singular). La continuación del string será "**a**" o "**an**", según el objeto empiece o no por vocal. Esta función está implementada en el método **get_number_objects**, que será explicado a continuación. El final del string es el nombre del objeto.

c) En caso de que la lista tenga más de un elemento, el comienzo del string se elegirá aleatoramiente (con la función **random.choice**) entre una de las frases que aparecen en verde:

Listado 5.51: Lista de posibles frases de descripción

```
1  possible_sentences = [" I can see ", " there are ", " there are some
       objects like ", " I can see these objects: "]
```

Todas ellas expresan la existencia de múltiples objetos.

A continuación se crea una lista vacía (**already_said**) para ir añadiendo los objetos que ya han sido incluidos en el string. Esto servirá para hacer una comprobación dentro del bucle que itera sobre la lista y no añadir un mismo objeto dos veces. Por ejemplo, serviría para evitar la siguiente situación:

I can see 2 books, a bowl, 2 books

Una vez realizada esta comprobación en el bucle, si el objeto todavía no ha sido añadido al texto, se calcula el número de veces que aparece mediante el método get_number_objects. Después, se comprueba si el número calculado es mayor que 1 para añadir o no la "**s**" al nombre y hacerlo plural. También se añade el objecto actual a la lista **already_said** para no repetirlo.

d) Se devuelve el string obtenido.

Esta es la función **get_number_objects** de la que se ha hablado anteriormente:

Listado 5.52: Método que devuelve el número de objetos en formato de texto

```
1  def get_number_objects(self, word, lst):
2          result = ""
3          num = lst.count(word)
4          if num == 1:
5                  if self.is_vowel(word[0]):
6                          result = "an "
7                  else:
8                          result = "a "
9          else:
10                 result = str(num) + " "
11         return result
```

El funcionamiento de esta función es muy sencillo. Se usa el método **count** para contar el número de apariencias de la palabra en la lista. Si el resultado es 1, se usa la función **is_vowel** para saber si el nombre del objeto comienza o no por

vocal y, por lo tanto, si se debe usar "**a**" o "**an**". Si el resultado es superior a 1, simplemente se devuelve el número convertido en string.

La implementación del método **is_vowel** es la siguiente:

Listado 5.53: Función que comprueba si una palabra empieza por vocal

```
def is_vowel(self, letter):
        if letter in self.vowels:
                return True
        return False
```

Simplemente se comprueba si la primera letra está contenida en la lista:

Listado 5.54: Lista de vocales

```
vowels = ['a', 'e', 'o', 'i', 'u']
```

En caso de que sí, se devuelve **True**. En caso contrario **False**.

5. Finalmente si imprimen las frases resultantes para comprobar si son correctas.

Funcionamiento

Para probar el descriptor de escenas, lo único que hay que hacer es ejecutar el nodo que publica los datos como se ha explicado anteriormente. Cuando este nodo esté activo y publicando datos obtenidos por YOLO, se debe abrir una nueva terminal e introducir los siguientes comandos para ejecutar este nuevo nodo:

Listado 5.55: Ejecución del descriptor de escenas

```
cd catkiwn_ws
source devel/setup.bash
rosrun tfg scene_descriptor.py
```

Este sería un posible resultado de los que se obtienen usando el **rosbag**:

```
alvaro@alvaro-Z270-HD3P:~/catkin_ws$ rosrun tfg scene_descriptor.py

SCENE DESCRIPTION
------------------------------------------------
I can see a cup
At the left side of the cup there are some objects like a mouse, 2 bottles, 2 remotes, a vase, a tvmonitor,
At the right side of the cup I can see a keyboard, a bottle, a laptop, 2 cups, 2 books,
------------------------------------------------
```

Figura 5.14: Ejemplo de ejecución del descriptor de escenas

5.3. Experimentación

En este apartado se van a comentar algunas pruebas que se han hecho para comprobar el correcto funcionamiento de todo lo que se ha explicado en los apartados anteriores. Estas pruebas consisten en verificar la robustez del sistema, de modo que se sepa a ciencia cierta su grado de fiabilidad.

5.3.1. Experimentos de detección de objetos

YOLO es una R-CNN que ha sido entrenada con imágenes de objetos reales. Por eso, es de suponer que no tendrá la misma efectividad a la hora de realizar predicciones sobre imágenes del mundo real que sobre modelos 3D de un mundo simulado. Por esta razón la sección se va a dividir en dos partes: experimentos usando Gazebo y experimentos usando imágenes reales. Después se compararán los resultados.

Experimentos en Gazebo

La detección correcta de objetos en Gazebo depende en gran medida de calidad de los modelos 3D. Esto quiere decir, que si los modelos reflejan la realidad decentemente, YOLO será capaz de predecir el tipo de objeto. En cambio, si los modelos no son muy realistas, es posible que la predicción sea errónea.

A continuación, pueden apreciarse dos ejemplos de prediccones erróneas por falta de realidad de los modelos. En el primero de ellos se intentaba reconocer una pelota (*cricket ball*, según el simulador). Por la falta de realismo, YOLO detecta el objeto como una naranja:

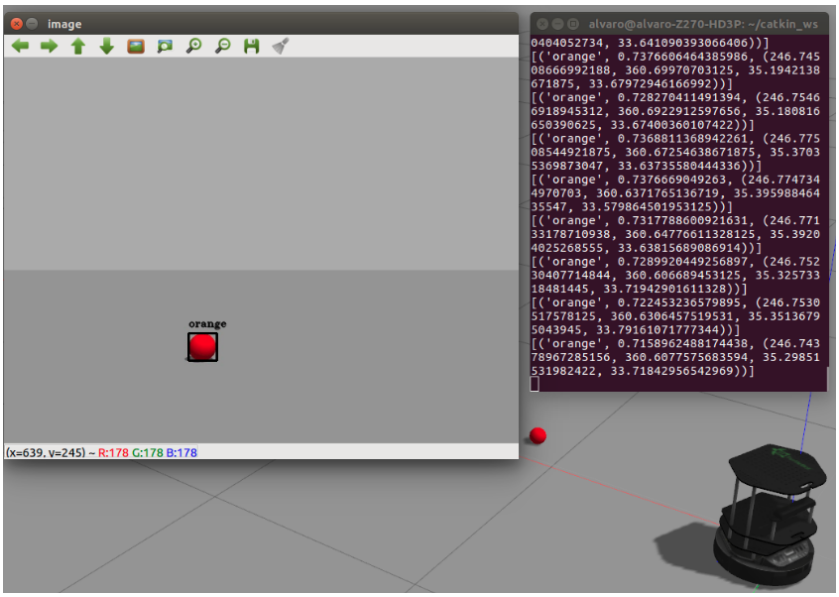


Figura 5.15: Detección errónea de una pelota en Gazebo

En el segundo de estos ejemplos, YOLO clasifica el objeto como una botella cuando en realidad es una lata:

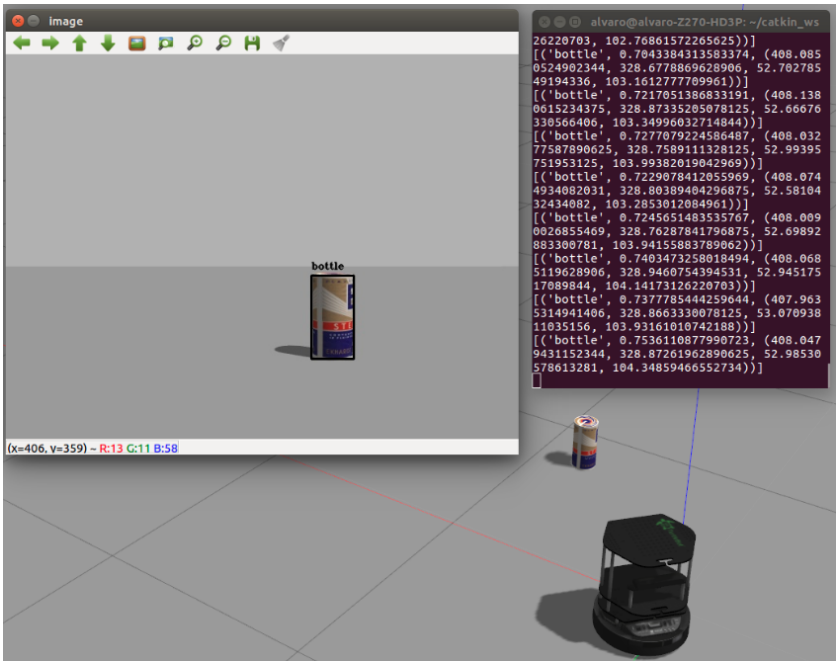


Figura 5.16: Detección errónea de una lata en Gazebo

En contraposición a estos casos, YOLO es capaz de reconocer correctamente el tipo de objeto cuando el modelo es realista. Aquí se muestran algunos ejemplos:

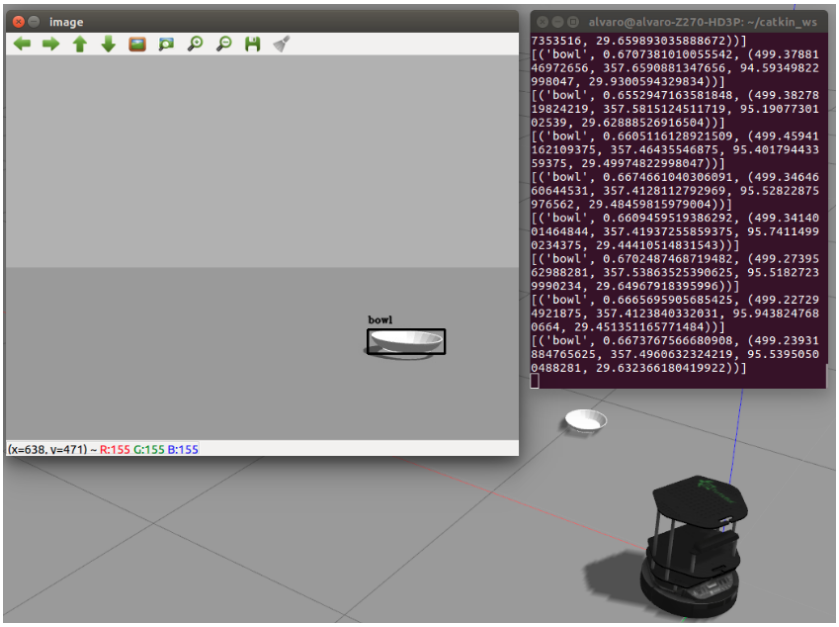


Figura 5.17: Detección correcta de un bol en Gazebo

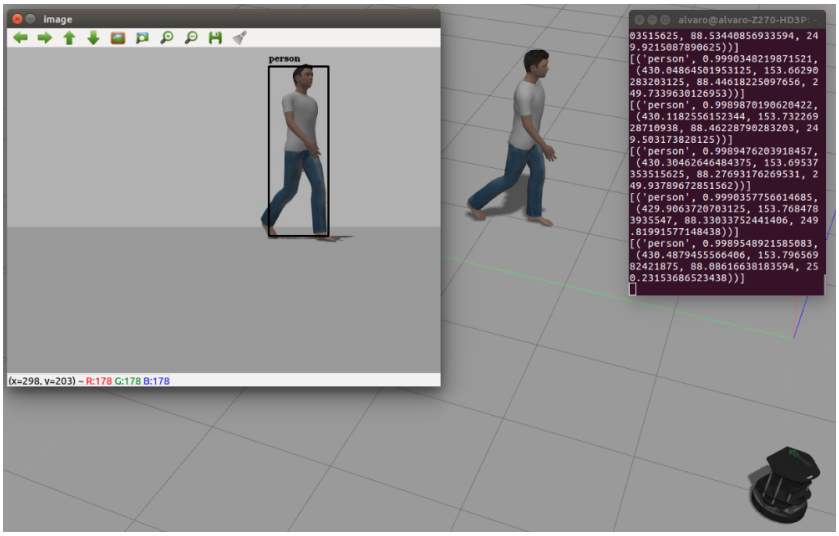


Figura 5.18: Detección correcta de una persona en Gazebo

5.3.2. Experimentos con objetos reales

A continuación se van a realizar predicciones sobre objetos reales utilizando YOLO con la webcam del ordenador. Se van a utilizar los mismos tipos de objetos que se han utilizado en Gazebo para poder comparar. Estos son los resultados obtenidos, en este caso YOLO no falla ninguna predicción:

Figura 5.19: Detección correcta de un bol real

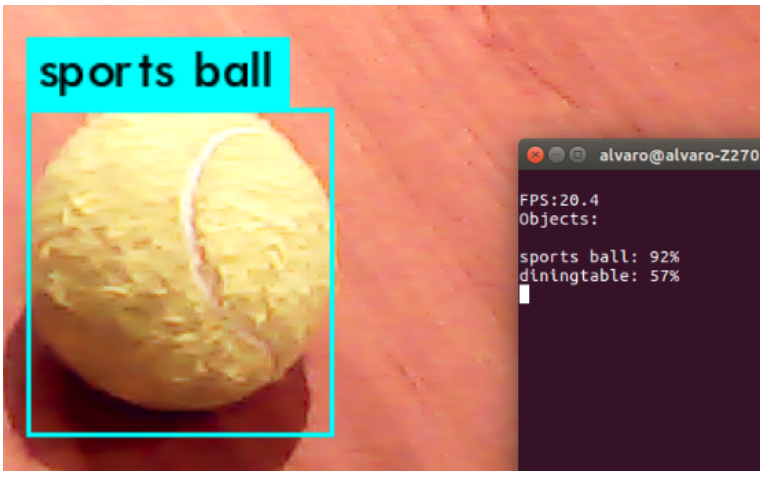


Figura 5.20: Detección correcta de una pelota real

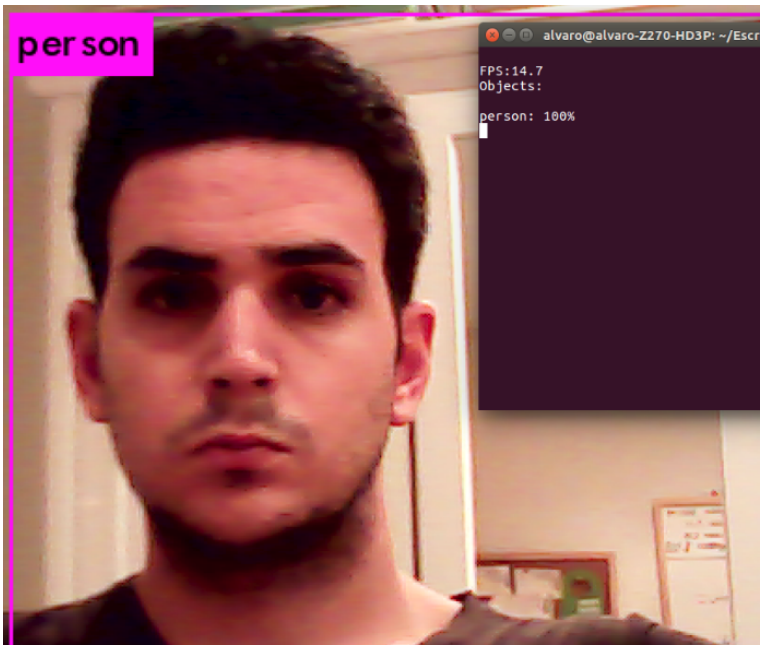


Figura 5.21: Detección correcta de una persona real

5.3.3. Comparación de resultados

La siguiente tabla sirve como comparatoria entre ambos experimentos y permite corroborar lo que se comentaba anteriormente. YOLO funciona mucho mejor en objetos reales que en modelos 3D, de hecho, en objetos reales tiene una tasa de fallo bastante baja.

OBJETO	GAZEBO	REALIDAD
Pelota	ERROR	92 %
Bol	66 %	96 %
Persona	99 %	100 %

Tabla 5.2: Comparativa de la precisión con la que se detecta cada objeto

Todos los datos expuestos en la tabla anterior han sido obtenidos mediante la realización de experimentos reales, esto puede ser comprobado mirando la imagen de cada experimento.

5.4. Experimentos en el laboratorio

Aquí se van a mostrar imágenes de experimentos realizados exitosamente en el laboratorio, con el Pepper real.

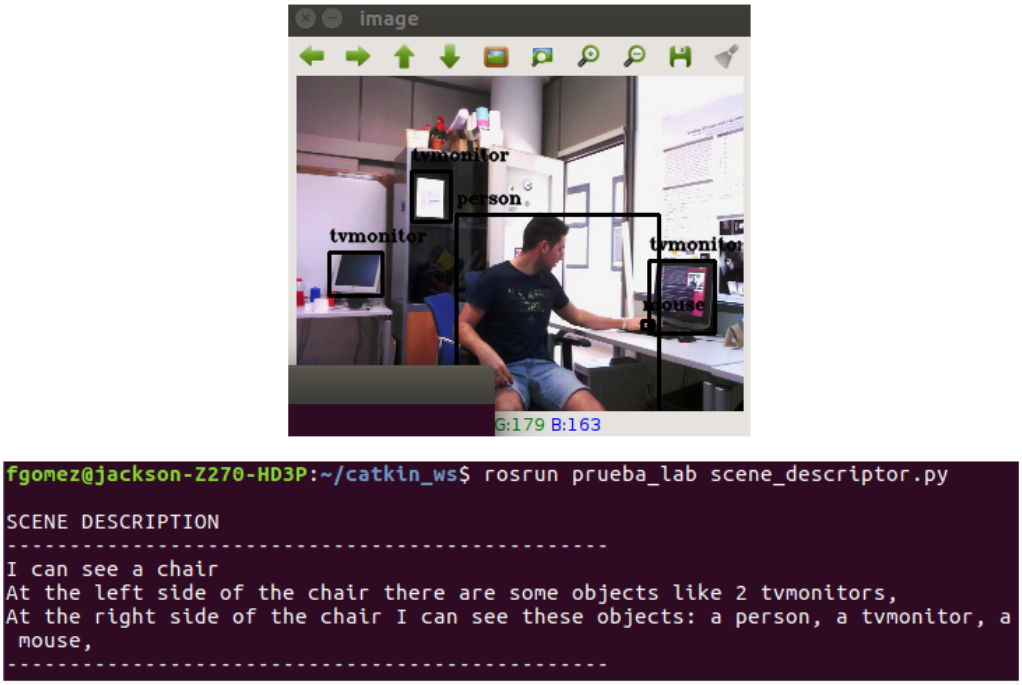
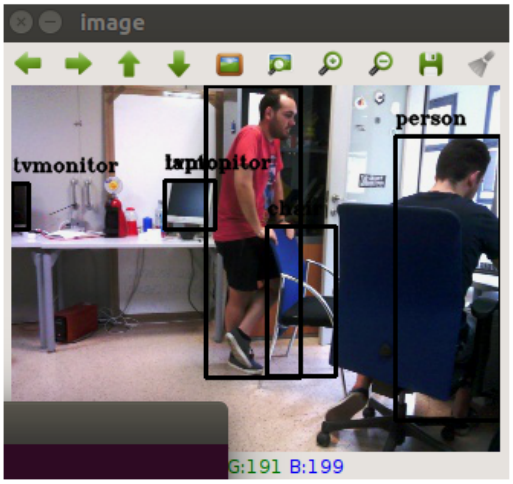


Figura 5.22: Primera prueba realizada en el laboratorio con Pepper

En esta imagen aparezco yo en el laboratorio ejecutando los nodos correspondientes para poner el sistema de descripción de escenas en marcha. La imagen ha sido obtenida de la cámara del robot Pepper. Como se puede observar, el robot es capaz de reconocer sin problema los objetos que hay por el laboratorio y de descrbir la escena que capta con su cámara.

Esta sería otra captura de pantalla de las que se tomó ese día:




```
fgomez@jackson-Z270-HD3P:~/catkin_ws$ rosrun prueba_lab scene_descriptor.py

SCENE DESCRIPTION
------------------------------------------------
I can see a laptop
At the left side of the laptop there is a tvmonitor
At the right side of the laptop there are 2 persons, a chair,
------------------------------------------------
```

Figura 5.23: Segunda prueba realizada en el laboratorio con Pepper

En esta ocasión, aparecemos mi tutor Francisco Gómez y yo realizando otra prueba para comprobar que la descripción de la escena es correcta.

Para finalizar, añado una última imágen:

Figura 5.24: Tercera prueba realizada en el laboratorio con Pepper

Aquí se puede ver otra vez a mi tutor rodeado de unos cuantos objetos que YOLO es capaz de reconocer.

6 Conclusiones

Para concluir este TFG se va a hacer un repaso de todo lo que se ha aprendido y conseguido. Además, se incluyen algunas reflexiones que pueden llegar a ser interesantes después de todo este proceso.

Estos son los resultados que se han conseguido en este proyecto expuestos de forma sintetizada:

- Se han realizado múltiples instalaciones de herramientas necesarias para poder llevar a cabo un proyecto de estas características.
- Se ha conseguido utilizar técnicas avanzadas de *Deep Learning* para reconocer objetos.
- Se ha integrado una R-CNN, como es YOLO, con ROS para conseguir detectar objetos utilizando robots.
- Se ha implementado un sistema de descripción de escenas utilizando datos posicionales de algunos objetos

A parte de conseguir resultados, el proyecto también perseguía algunos objetivos que se han tenido que ir cumpliendo poco a poco para poder llegar a este punto. Estos objetivos no son tan concretos, como la implementación de un descriptor de escenas o el reconocimiento de objetos, sino que son algo más abstractos. Son tareas que se han ido completando "en segundo plano", a la vez que las implementaciones. Los objetivos cumplidos son los siguientes:

- Se ha aprendido a utilizar ROS, y por lo tanto, a trabajar y a desarrollar aplicaciones para robots.
- Se ha investigado sobre diversos métodos de reconocimiento de objetos.
- Se ha aprendido sobre algunas técnicas de Inteligencia Artificial (IA), como *Machine Learning* y *Deep Learning*.

Para resumir, se podría decir que mediante el uso de técnicas muy modernas se ha podido resolver el problema planteado incialmente. Me gustaría destacar que sin el uso de algunos de estos métodos y herramientas, como el *Deep Learning* o el framework ROS entre otras, este mismo proyecto hubiese sido mucho más complejo de llevar a cabo. Es por eso, que todo esto puede servir como ejemplo para demostrar lo mucho que se está

avanzando hoy en día y las barrreras que es capaz de superar el ser humano.

Dejando de lado el proyecto en cuestión, me gustaría destacar otro detalle. Durante la realización de este grado se han realizado muchos proyectos y prácticas, pero la mayoría de ellos han sido muy guiados y trabajando en grupo. Sin embargo, al enfrentarme a un proyecto como este, en el que he tenido que afrontar muchos problemas, he aprendido a ser más independiente a la hora de trabajar y a resolver situaciones por mi cuenta. Sí que es cierto que los tutores suponen un apoyo muy grande y que siempre que han podido me han ayudado, pero al final en un proyecto de estas características es uno mismo quien tiene que plantear sus propias soluciones y ponerse sus límites.

Lista de Acrónimos

TFG: Trabajo Final de Grado
ROS: *Robot Operating System*
IA: Inteligencia Artifical
YOLO: *You Only Look Once*
CUDA: *Compute Unified Device Architecture*
GPU: *Graphics Processing Unit*
CPU: *Central Processing Unit*
CNN: *Convolutional Neural Network*
R-CNN: *Region based Convolutional Neural Network*
GB: *Gigabyte*
Ah: Amperios-hora
Wh: Vatios-hora

Bibliografía

[man, 2015] (2015). Manual de ROS. https://moodle2015-16.ua.es/moodle/mod/book/view.php?id=82546.

[nvi, 2018] (2018). Comprobación de la versión de *Nvidia Drivers*. https://www.nvidia.es/Download/index.aspx?lang=es.

[dem, 2018] (2018). Demostración del funcionamiento de YOLO. https://youtu.be/MPU2HistivI.

[CNN, 2018] (2018). Explicación de las CNN. https://ch.mathworks.com/solutions/deep-learning/convolutional-neural-network.html.

[rec, 2018] (2018). Investigación sobre reconocimiento de objetos. https://ch.mathworks.com/solutions/deep-learning/object-recognition.html.

[YOL, 2018] (2018). Manual de YOLO. https://pjreddie.com/darknet/yolo/.

[Mat, 2018] (2018). Math works. Disponible en https://ch.mathworks.com/.

[ROS, 2018] (2018). Página oficial de ROS. http://www.ros.org/.

[Pep, 2018] (2018). Página oficial de aliverobots, distribuidores de pepper. https://aliverobots.com/robot-pepper/.

[Dar, 2018] (2018). Página oficial de *Darknet*. https://pjreddie.com/darknet/.

[DL, 2018] (2018). Teoría de *Deep Learning*. https://ch.mathworks.com/discovery/deep-learning.html.

[ML, 2018] (2018). Teoría de *Machine Learning*. https://ch.mathworks.com/discovery/machine-learning.html.